

2014

Effective Resource and Workload Management in Data Centers

Lei Lu

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

Recommended Citation

Lu, Lei, "Effective Resource and Workload Management in Data Centers" (2014). *Dissertations, Theses, and Masters Projects*. Paper 1539623637.

<https://dx.doi.org/doi:10.21220/s2-xfd5-y603>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Effective Resource and Workload Management in Data Centers

Lei Lu

Tianjin, China

**Bachelor of Science, Nanjing University, 2005
Master of Engineering, Nanjing University, 2008**

**A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy**

Department of Computer Science

**The College of William and Mary
January, 2014**

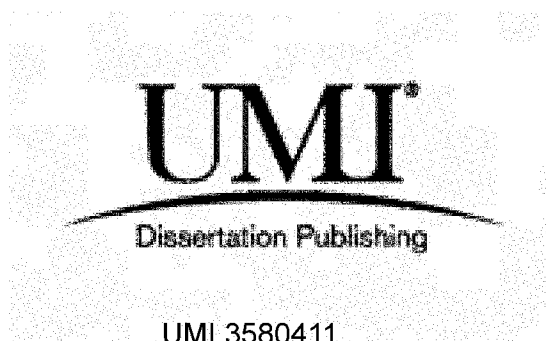
UMI Number: 3580411

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 3580411

Published by ProQuest LLC 2014. Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

APPROVAL PAGE

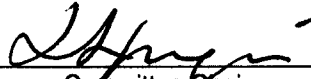
This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy



Lei Lu

Approved by the Committee, October 2013



Committee Chair

Professor Evgenia Smirni, Computer Science
The College of William and Mary



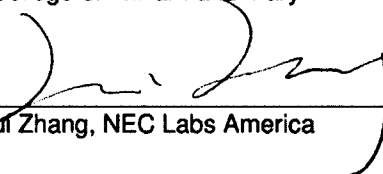
Professor Weizhen Mao, Computer Science
The College of William and Mary



Associate Professor Peter Kemper, Computer Science
The College of William and Mary



Associate Professor Haining Wang, Computer Science
The College of William and Mary



Dr. Hui Zhang, NEC Labs America

ABSTRACT

The increasing demand for storage, computation, and business continuity has driven the growth of data centers. Managing data centers efficiently is a difficult task because of the wide variety of datacenter applications, their ever-changing intensities, and the fact that application performance targets may differ widely. Server virtualization has been a game-changing technology for IT, providing the possibility to support multiple virtual machines (VMs) simultaneously. This dissertation focuses on how virtualization technologies can be utilized to develop new tools for maintaining high resource utilization, for achieving high application performance, and for reducing the cost of data center management.

For multi-tiered applications, bursty workload traffic can significantly deteriorate performance. This dissertation proposes an admission control algorithm AWAIT, for handling overloading conditions in multi-tier web services. AWAIT places on hold requests of accepted sessions and refuses to admit new sessions when the system is in a sudden workload surge. To meet the service-level objective, AWAIT serves the requests in the blocking queue with high priority. The size of the queue is dynamically determined according to the workload burstiness.

Many admission control policies are triggered by instantaneous measurements of system resource usage, e.g., CPU utilization. This dissertation first demonstrates that directly measuring virtual machine resource utilizations with standard tools cannot always lead to accurate estimates. A directed factor graph (DFG) model is defined to model the dependencies among multiple types of resources across physical and virtual layers.

Virtualized data centers always enable sharing of resources among hosted applications for achieving high resource utilization. However, it is difficult to satisfy application SLOs on a shared infrastructure, as application workload patterns change over time. AppRM, an automated management system not only allocates right amount of resources to applications for their performance target but also adjusts to dynamic workloads using an adaptive model.

Server consolidation is one of the key applications of server virtualization. This dissertation proposes a VM consolidation mechanism, first by extending the fair load balancing scheme for multi-dimensional vector scheduling, and then by using a queueing network model to capture the service contentions for a particular virtual machine placement.

TABLE OF CONTENTS

| | |
|--|------|
| Acknowledgments | vi |
| Dedication | vii |
| List of Tables | viii |
| List of Figures | ix |
| 1 Introduction | 1 |
| 1.1 Overview of Data Centers | 1 |
| 1.2 Dissertation Contributions | 3 |
| 1.2.1 Summary of Contributions | 3 |
| 1.2.2 Management of Application Workloads | 4 |
| 1.2.3 Virtualized Server Measurement | 5 |
| 1.2.4 Autonomic Resource Control | 5 |
| 1.2.5 Virtual Machine Consolidation Strategy | 6 |
| 1.3 Organization | 7 |
| 2 Background and Related Work | 8 |
| 2.1 Application Architecture Overview | 8 |
| 2.1.1 The Client/Server and n-Tier Models | 8 |
| 2.1.2 Admission Control in Web Servers | 9 |
| 2.1.3 Related Work on Admission Control | 10 |
| 2.2 Virtualization of Data Centers | 13 |
| 2.2.1 Background | 13 |
| 2.2.2 Virtualization Conditions | 14 |

| | | |
|---------|--|----|
| 2.3 | Server Consolidation | 17 |
| 2.3.1 | Virtual Machine Migration | 17 |
| 2.3.2 | Performance Isolation Among Virtual Machines | 18 |
| 2.3.3 | Related Work on Consolidation | 19 |
| 2.3.3.1 | Measurement and Characterization | 20 |
| 2.3.3.2 | Consolidation Strategies | 22 |
| 2.4 | Feedback Control-based Resource Allocation | 24 |
| 3 | Admission Control for Busy Multi-tier Services | 27 |
| 3.1 | Motivation: Capacity Planning and Admission Control | 30 |
| 3.1.1 | Basic Capacity Planning | 30 |
| 3.1.2 | Introducing Burstiness | 33 |
| 3.1.3 | Burstiness in Flows and Admission Control | 35 |
| 3.2 | AWAIT Algorithm | 37 |
| 3.2.1 | Static AWAIT | 38 |
| 3.2.2 | Performance Evaluation: Conservative or Aggressive? | 43 |
| 3.2.3 | Performance Effect of the Blocking Queue Size | 46 |
| 3.2.4 | Handle the Effects of Bottleneck Switch | 47 |
| 3.3 | Autonomic AWAIT | 49 |
| 3.4 | Comparisons with an Approach Based on Control Theory | 56 |
| 3.5 | Summary | 59 |
| 4 | Calibrating Resource Utilization in VMs | 60 |
| 4.1 | Problem Formulation | 62 |
| 4.1.1 | Xen Virtualization | 62 |
| 4.1.2 | Information Mismatching Paradox | 63 |
| 4.1.3 | Problem Formulation | 64 |
| 4.1.3.1 | Virtual Resource Monitored Information | 65 |

| | |
|---|----|
| 4.1.3.2 Physical Resource Monitored Information | 66 |
| 4.2 Background Information | 66 |
| 4.2.1 Source Separation | 66 |
| 4.2.2 Factor Graphs | 67 |
| 4.3 Directed Factor Graphs | 68 |
| 4.3.1 Graph Model | 68 |
| 4.3.2 DFG in VM Information Calibration | 69 |
| 4.4 DFG Based Model | 70 |
| 4.4.1 Methodology | 70 |
| 4.4.2 Regression Analysis | 71 |
| 4.4.2.1 Source Node: Virtual CPU Load | 71 |
| 4.4.2.2 Source Node: Virtual Memory Load | 71 |
| 4.4.2.3 Source Node: Virtual Network Load | 72 |
| 4.4.2.4 Source Node: Virtual Disk IO Load | 73 |
| 4.4.2.5 System Overhead | 75 |
| 4.5 Information Calibration | 75 |
| 4.5.1 Run-time Calibration Mechanism | 75 |
| 4.5.2 Robust Remodeling: Guided Regression | 77 |
| 4.6 Evaluation | 80 |
| 4.6.1 Experimental Methodology | 80 |
| 4.6.2 Results | 81 |
| 4.6.2.1 Scenario 1: RUBiS | 82 |
| 4.6.2.2 Scenario 2: Co-hosting Network- and IO-intensive Apps | 83 |
| 4.6.2.3 Scenario 3: Co-hosting IO-intensive Apps | 85 |
| 4.7 Summary | 85 |
| 5 Auto-Scaling of VMs in Resource Pools | 87 |
| 5.1 Architecture | 90 |

| | |
|---|-----|
| 5.2 Design | 92 |
| 5.2.1 Sensors | 92 |
| 5.2.2 Model Builder | 93 |
| 5.2.3 Application Controller | 94 |
| 5.2.4 Resource Controller | 96 |
| 5.2.5 Resource Pool (RP) Manager | 97 |
| 5.3 Testbed Setup | 101 |
| 5.4 Performance Evaluation | 102 |
| 5.4.1 Achieving Performance Targets for Multiple Metrics | 103 |
| 5.4.2 Detecting and Mitigating Dynamically-Changing Workload De- mands | 106 |
| 5.4.3 Applying Control on Multiple Applications | 108 |
| 5.4.4 Enforcing Performance Targets under Competing Workloads . . | 110 |
| 5.4.4.1 Non-expandable and Unmodifiable Resource Pool . . . | 110 |
| 5.4.4.2 Non-expandable but Modifiable Resource Pool | 111 |
| 5.4.4.3 Expandable Resource Pool | 115 |
| 5.5 Summary | 116 |
| 6 Predictive VM Consolidation | 117 |
| 6.1 Background: Fair Load Balancing on a Single Resource | 119 |
| 6.1.1 Max-Min Fairness | 120 |
| 6.1.2 Min-Max Load Balancing | 120 |
| 6.1.3 Fair Load Balancing | 120 |
| 6.2 Fair Load Balancing on Multiple Resources: Challenges | 121 |
| 6.3 Fair VM Allocation Algorithm | 123 |
| 6.3.1 Vector Bin Packing | 123 |
| 6.3.2 Discussion | 126 |
| 6.3.3 Vector Scheduling with Predictive Model | 127 |

| | |
|--|-----|
| 6.4 A Predictive Queueing Model | 129 |
| 6.4.1 RUBiS Multi-tiered Benchmark | 129 |
| 6.4.2 Application Profiling: Service Demand Estimation | 130 |
| 6.4.2.1 Assumptions | 131 |
| 6.4.2.2 Per-tier Service Time | 131 |
| 6.4.2.3 Average Network Demand | 132 |
| 6.4.2.4 Average Disk Demand | 133 |
| 6.4.2.5 Average CPU Demand | 134 |
| 6.5 Evaluation | 134 |
| 6.5.1 RUBiS with Different Number of Clients | 136 |
| 6.5.2 Browsing and Bidding Mix Consolidation | 136 |
| 6.5.3 Browsing Mix Consolidation | 140 |
| 6.6 Summary | 143 |
| 7 Summary of Contributions and Future Work | 144 |
| 7.1 Future Work | 146 |
| 7.1.1 Autonomic Resource Allocation | 146 |
| 7.1.2 VM Auto-scaling | 148 |
| 7.1.3 Predictive Server Consolidation in Multi-cores | 148 |
| 7.1.4 Server Consolidation with Performance Target | 149 |
| A Markovian Arrival Processes | 151 |
| References | 154 |
| VITA | 172 |

ACKNOWLEDGMENTS

This dissertation would not have been accomplished without the guidance of my advisor and the support of many people. First and foremost, I want to express my deep and sincere gratitude to my advisor Professor Evgenia Smirni. Her work ethic and commitment to the high quality research have set a model for me to follow. In the long journey to my doctoral degree, she taught me the key knowledge and skills to be competent for a researcher. I am grateful for the freedom and trust she gave me to explore my own research interests, the guidance and encouragement whenever I needed it. It has been my great honor to be one of her students.

I would like to thank Hui Zhang, my mentor at NEC Labs. He gave me the opportunity to do research in an industry environment and introduced me to performance research in virtualization. He spent countless hours giving me invaluable advice and collaborating on the research projects. I am also grateful to Xiaoyun Zhu, my mentor at VMware. She guided me patiently through my internship and explained the details of theory knowledge to me. She also gave me valuable advice and firm support in my career development.

I have collaborated with researchers outside of W&M. I thank Lucy Cherkasova, Vittoria De Nitto Personé, and Ningfang Mi for their invaluable feedback during our close collaboration. Guofei Jiang, Kenji Yoshihira, and Haifeng Chen of NEC Labs collaborated on my project in NEC Labs. Rean Griffith, Pradeep Padala, Aashish Parikh, and Parth Shah in VMware helped in completing the AppRM project.

I would also like to thank my committee members, Professors Weizhen Mao, Peter Kemper, and Haining Wang for their valuable feedback and suggestions that beneficially helped me improve this dissertation. I cordially thank Professors Gang Zhou, Andreas Stathopoulos, Phil Kearns, and Qun Li for their superb teaching. In addition, I would like to deeply thank Vanessa Godwin and Jacqlyn Johnson for their direction on many administrative works over the years.

My peers in the Computer Science department played an important role during my Ph.D. time. Andrew Caniff answered my questions and assisted with my first project. Feng Yan provided information and discussions whenever I needed. Thanks also go to Zhen Ren, Xin Qi, Bo Dong, Fengyuan Xu, Zhenyu Wu, Chuan Yue, Ruth Lamprecht, Jidong Xiao, Zhijia Zhao, and too many others to list who helped me and made my time in Williamsburg more enjoyable.

I would never come this far without the unconditional love from my family. My parents have always been supporting, understanding, and encouraging me along the way. Finally, a very special thanks goes to my wife, Haiyan Zhu. She walked together with me through my happy times and hard times over years. I look forward to continuing our journey together.

To my wife and my parents,
for their dedicated love and continued support ...

LIST OF TABLES

| | | |
|-----|---|-----|
| 3.1 | Configuration | 34 |
| 4.1 | Network regression model. | 73 |
| 4.2 | Blktap based device regression model | 74 |
| 4.3 | Loopback based device regression model | 74 |
| 4.4 | Example of DFG error that triggers remodeling mechanism | 84 |
| 5.1 | Notation | 94 |
| 5.2 | Configuration of hosts | 102 |
| 5.3 | Definition of three changing workloads | 106 |
| 6.1 | Profiling of browsing and bidding workload | 135 |
| 6.2 | Configuration table | 137 |
| 6.3 | Service loads for three workloads | 138 |
| 6.4 | Min-max load balancing for the bidding and browsing mix consolidation | 138 |
| 6.5 | Predicted performance for min-max configs | 139 |
| 6.6 | Min-max load balancing for browsing mixes consolidation | 141 |
| 6.7 | Predicted performance for min-max configs | 141 |

LIST OF FIGURES

| | | |
|-----|--|----|
| 2.1 | The Client/Server and n-Tier Model | 9 |
| 3.1 | The basic structure of TPC-W model | 31 |
| 3.2 | Capacity Planning study for SBAC under exponential (i.e., not bursty) new session arrivals. Performance measures are presented as a function of the maximum number of active requests in the system. . | 33 |
| 3.3 | The burstiness profiles of the three arrival MAPs. | 34 |
| 3.4 | Three different burstiness profiles. The capacity planning results and SLO targets are now violated. It appears that a queue size of 256 (i.e., maximum active requests for the baseAC configuration) is not sufficient to meet SLO requirements. | 36 |
| 3.5 | The model of AWAIT algorithm | 37 |
| 3.6 | AWAIT with fixed size of the blocking queue. The graphs illustrate performance values for the aggressive and conservative versions (see white and shaded bars, respectively) for various fixed sizes of the blocking queue B . In all experiments, the limit of accepted requests A is set to 256, based on capacity planning. | 44 |
| 3.7 | CCDF of system response time for different strategies for aggressive AWAIT that operates with a blocking queue size of 16 slots and a blocking queue size has 128 slots. | 46 |
| 3.8 | SBAC performance under the three different burstiness profiles with highly overloaded back-end server. | 48 |

| | | |
|------|--|----|
| 3.9 | Aggressive AWAIT: strategy performance for various fixed sizes of the blocking queue B with highly overloaded back-end server. In the first row, the horizontal lines inside the bars reflect the average request processing time. | 50 |
| 3.10 | Autonomic AWAIT: illustration of how the capacity of the blocking queue B changes as a function of the workload. | 53 |
| 3.11 | (a) Arrival process with different burstiness levels; (b) Blocking queue capacity changes as a function of the workload. | 54 |
| 3.12 | Moving 95th percentile of response time and moving average of aborted ratio, drop ratio and completed session ratio under the request arrival pattern shown in Figure 3.11 (a). | 56 |
| 3.13 | Comparison of performance of autonomic AWAIT and the control theory-based algorithm developed in [29]. | 58 |
| 4.1 | Measurement information mismatching: a disk I/O utilization example | 63 |
| 4.2 | Measurement information mismatching: a CPU utilization example . | 64 |
| 4.3 | An example factor graph | 67 |
| 4.4 | A directed factor graph example | 68 |
| 4.5 | The directed factor graph in VM monitoring information calibration. . | 69 |
| 4.6 | Compute intensive workload has no impact on privileged domain performance | 72 |
| 4.7 | The run-time calibration mechanism | 75 |
| 4.8 | Relative error of in-VM monitoring method and DFG based model in RUBiS app | 83 |
| 4.9 | Relative error for in-VM monitoring method and run time calibration mechanism | 85 |
| 4.10 | Relative error for in-VM monitoring and DFG model in mixed signal decomposing | 86 |

| | | |
|------|---|-----|
| 5.1 | An example VDC containing two resource pools hosting two multi-tier vApps and two single-tier VMs. | 88 |
| 5.2 | AppRM at work across VMs in a single vApp. | 91 |
| 5.3 | AppRM at work across vApps in a resource pool. | 92 |
| 5.4 | Experimental setup with a Mongodb cluster and Rain benchmark . . | 103 |
| 5.5 | Mean response time target (300 ms) | 104 |
| 5.6 | Resource utilization for under-provisioning | 104 |
| 5.7 | Resource utilization for over-provisioning | 105 |
| 5.8 | 95 th percentile response time target (2000 ms) | 105 |
| 5.9 | Throughput target (50,000 reqs/s) | 105 |
| 5.10 | Measured performance under dynamic workloads with target 500 ms | 107 |
| 5.11 | Measured performance under dynamic workloads with target 600 ms | 108 |
| 5.12 | Resource utilization for dynamic workloads | 108 |
| 5.13 | Measured performance under dynamic workloads with target 800 ms | 109 |
| 5.14 | Resource utilization for dynamic workloads | 109 |
| 5.15 | Application 1 mean target 1000 ms and Application 2 target 600 ms | 109 |
| 5.16 | Testbed setup for resource pool experiments | 110 |
| 5.17 | Measured application relative performance in non-expandable and unmodifiable RP (targets: 600ms) | 111 |
| 5.18 | Measured application relative performance in non-expandable and modifiable RP (targets 600ms) | 112 |
| 5.19 | Application and RP level reservations | 112 |
| 5.20 | Measured application relative performance in non-expandable and modifiable RP (targets 600ms) | 113 |
| 5.21 | RP level reservation | 113 |
| 5.22 | Measured application relative performance in non-expandable and modifiable RP (targets 600ms) | 114 |

| | | |
|------|--|-----|
| 5.23 | RP level reservation setting | 114 |
| 5.24 | Measured application relative performance in expandable RP (targets 600ms) | 115 |
| 5.25 | Application and RP level reservations | 115 |
| 6.1 | A simple load balancing problem on two resources (CPU, memory). | 122 |
| 6.2 | Closed queueing network model for multi-tier server and multi-class workload | 130 |
| 6.3 | An example of tcprstat query result | 132 |
| 6.4 | PREMATCH architecture graph. | 135 |
| 6.5 | Measured vs. predicted average response time and throughput. | 136 |
| 6.6 | Measured vs. predicted average response time for browsing and bidding mix. | 139 |
| 6.7 | PREMATCH, random, and worst consolidation comparison. | 140 |
| 6.8 | Measured vs. predicted average response time for the browsing mixes. | 142 |
| 6.9 | PREMATCH, random, and worst consolidation comparison. | 142 |

1 Introduction

1.1 Overview of Data Centers

Data centers are rapidly becoming the standard IT solution to host internet and businesses applications due to their great potential in providing highly reliable service and reducing operation cost. According to a definition from Wikipedia [25], Data Center is

“a facility used to house computer systems and associated components, such as telecommunications and storage systems. It generally includes redundant or backup power supplies, redundant data communications connections, environmental controls (e.g., air conditioning, fire suppression) and security devices. ”

It is reported that more than one million servers are scattered in three dozen data centers around the world [6]. Effective management of resources in such environment results in many challenging and interesting research problems.

Virtualization brings dramatic changes in data centers. It enables partitioning a single physical server into multiple virtual machines, each with its own independent application and operating system. Instead of provisioning a single, physical server with enough spare (often idle) capacity to support the peak load of a single application, virtualization provides a way to isolate and partition server resources to meet the variable demands of application workloads. The trend to use server

virtualization technologies to consolidate multiple data center servers is growing rapidly. It is reported that 48% percent of x86 server OS instances are operated as virtual servers by 2012 and this number is expected to grow by 74% over the next two years [19].

Server virtualization enables the rapid and fine-grained resource management in server systems. Enterprise virtualization products allow dynamically adjusting CPU and memory resources while the virtual machine (VM) is running. However, as data centers continue to deploy virtualized solutions, new challenges have also emerged: accurately monitoring virtualized applications demand; correct resource allocation for meeting performance goals; determining optimal VM placements strategies to reduce the workload interference, are some examples of pressing challenges.

This dissertation aims at developing a systematic methodology that allows for improved solutions when dealing with challenges related to virtualization overhead measurement, autonomic resource management, and optimized VM placement. This dissertation provides answers to the following questions:

- How can we effectively design an admission control policy for prevalent applications used by data center tenants?
- How can we estimate accurate resource utilizations of an application running in a virtualized environment when we need to consider virtualization overheads?
- How can we automatically and efficiently set the resource controls for VMs and resource pools to meet the applications SLOs? How can the system ensure performance to individual application in spite of dynamically-changing workloads?
- What is the best way to colocate applications, i.e., what are the workload

characteristics of competing applications that are best to be matched in order to obtain an optimal workload "mix" such that performance interference is minimized?

The above questions cover many of resource and workload management problems from optimizing workloads placement to system monitoring and performance enforcement. Under the requirement of rapid response and scalability for data centers, the questions become more challenging. If these questions are solved, data centers can be made more efficient, autonomic, and significantly cost effective in management.

1.2 Dissertation Contributions

This dissertation mainly focuses on providing automated solutions to system monitoring and resource management challenges that data centers face today. Techniques that combine virtualization with intelligent control algorithms and system modeling are developed in this dissertation.

1.2.1 Summary of Contributions

Overall, the key contributions of this dissertation are:

- **AWAIT**: a novel admission control policy that utilizes the concept of blocking queue as an overload protection mechanism for bursty workloads [90],
- **DFG**: an automated technique that quantifies the cost of virtualization layer overheads to accurately calibrate VM resource demands [91],
- **AppRM**: a performance management tool that automatically adjusts resource control settings at the individual virtual machine level or at the resource pool

level such that the virtualized applications running in a virtual data center can meet their respective performance goals [93], and

- **PREMATCH**: an automated placement engine that provides multi-dimensional min-max load balancing and minimizes interference among co-located VMs [92].

1.2.2 Management of Application Workloads

Multi-tier web service is a popular application paradigm in data centers. Meeting SLOs in web services is a challenging and complex problem. While capacity planning is widely used to size the system and meet SLOs under normal operating conditions, it is exceedingly difficult to effectively meet performance and operation targets when web traffic conditions become bursty. The deficiency of well-accepted techniques for admission control for single-tiered systems when applied in the prevalent multi-tier setting under bursty conditions has been documented [98]. Such policies unavoidably result in rejecting requests of already accepted sessions, which directly translate into significant business loss.

To remedy this problem, a novel autonomic admission control policy, called *AWAIT* is proposed. It utilizes the concept of "blocking queue" as an overload protection mechanism. When the system experiences sudden overload and starts operating above capacity, requests from accepted sessions are not aborted but are instead stored in a blocking queue that effectively operates like a "waiting room" but with the unavoidable caveat of jeopardizing the targeted request latency due to the extra waiting. After overload subsides, requests in the blocking queue are served with high priority. *AWAIT* effectively adjusts the size of the blocking queue in an autonomic way and strikes a good balance among two conflicting goals: restricting the size of the blocking queue to best meet target SLOs, while continuously adapting its size in order to best react to workload burstiness.

1.2.3 Virtualized Server Measurement

Server virtualization brings benefits in autonomic resource management, but also leads to new challenges. The challenge addressed in this dissertation is on profiling physical resource utilization information of VMs when consolidated on a single server. Profiling is very difficult due to dynamic mapping relationships of resource activities between the virtual layer and the physical layer. The problem is further exacerbated by cross-resource utilization causality relationships due to virtualization overhead and resource utilization multiplexing across different VMs.

The profiling problem is formulated as a source separation problem as studied in digital signal processing and uses a directed factor graph (DFG) to model the multivariate dependence relationships among different resources (CPU, memory, disk, network) across virtual and physical layers. A benchmark-based methodology is designed to build a DFG based model for the VM information calibration problem. A run-time calibration mechanism is proposed based on the DFG based model and further enhanced with a robust remodeling method based on guided regression. The proposed methodology outputs estimates of physical resource utilization on individual VMs and physical server aggregate resource utilization.

1.2.4 Autonomic Resource Control

Virtual data centers (VDC) and Resource pools (RPs) are logical containers representing an aggregate resource allocation for a collection of virtual machines being managed by VMware's cloud management software. Resource pools offer powerful resource control primitives including reservations, limits, and shares that can be set at a VM or at a resource pool level. These primitives allow administrators to control the absolute and relative amount of resources a VM or a resource pool consumes. However, as the virtual machine sprawl continues, it has become in-

creasingly difficult to set these knobs properly such that virtualized applications (referred to as vApps) can get enough resources to meet their respective SLOs.

In this dissertation, a tool called *AppRM* is presented. It is able to automatically set the resource controls for VMs and resource pools to meet the application SLOs. *AppRM* contains a hierarchy of vApp Managers and RP Managers, where a vApp Manager translates the SLO for an application into the resource control settings for the individual VMs running that application. An RP Manager ensures that all applications within the resource pool can meet their SLOs by adjusting the knobs at the resource pool level. Each vApp Manager consists of a model builder, an application controller, and a resource controller.

1.2.5 Virtual Machine Consolidation Strategy

Effective consolidation of different applications on common resources is often akin to a black art as unexpected application performance interference may result in unpredictable system and workload delays. The problem of fair load balancing on multiple servers within a virtualized data center setting is addressed in this dissertation. Especially it is focused on multi-tiered applications with different resource demands per tier and address the problem on how to best match each application tier on each resource such that performance interference is minimized.

For this specific problem, a two-step approach is proposed. First, a load balancing algorithm is developed that assigns different virtual machines across different servers by applying min-max load balancing on individual server loads, aiming at balancing the load across all servers. This process is formulated as a multi-dimensional vector scheduling problem that uses a polynomial time approximation scheme to minimize the maximum utilization across all server resources and results in several load balancing solutions. As a second step, a queueing network analytic model is applied on the proposed min-max solutions. The model predicts

the application performance under multiple consolidation choices and selects the optimal balancing solution.

1.3 Organization

The rest of this document is organized as follows. Chapter 2 gives background and related work on virtualized data centers to set the context of this work. Chapter 3 describes an admission control policy for handling overloading conditions in multi-tier web services. This is followed in Chapter 4 with a discussion of how to quantify the cost of virtualization layer overheads in order to calibrate measured VM resource demand. Chapter 5 discusses a performance management tool that automatically adjusts resource control settings at individual virtual machine levels to allow virtualized applications meeting their respective performance goals. Chapter 6 proposes how to co-locate multi-tiered applications on a given set of physical resources in a multi-tenant data center. Finally, in Chapter 7 gives a summary of contribution and outlines future work.

2 Background and Related Work

This chapter presents the background material and detailed related work that puts the contribution of this dissertation into perspective.

2.1 Application Architecture Overview

The main purpose of data centers is hosting and running the core business application environment of corporations. Most of today's enterprise applications use a web-based front end. Since a successful design of resource and workload management policy requires good understanding of the application characteristics, this section provides a high-level overview of today's application architecture.

2.1.1 The Client/Server and n-Tier Models

Most applications today are developed according to the client/server or n-tier models. In fact, for most enterprise software, the client/server model has evolved to the n-tier model. The client/server model was originated from Xerox PARC during the 1970s [2]. In this architecture, as shown in Figure 2.1(a), the client application is a part of the application program running at the client's computer to retrieve data from the server and present it to the user. The server, most commonly a database management system, stores application data, such as user information. The presentation, business logic, and data provision are separated in the n-tier model to

minimize the impact of logic changes, see Figure 2.1(b). According to this model, the application functions are divided into the following software tiers:

- The client tier -- The client software (usually a web browser) renders the user interface.
- The presentation tier -- This software provides the function of user interface generation. It comprises *static objects*, such as images, and *dynamically generated objects* to translate the results of the application computation to something that the user can understand. On web-based applications, the presentation tier is implemented by web servers.
- The application tier -- This tier provides the business logic, coordinates the application, and performs calculations. The application tier typically connects the presentation tier and database tier. It receives remote procedure calls from the presentation tier, stores and retrieves data from the database, and returns the result of the computation to the presentation tier. The typical technologies are ASP, Java servlets, and EJB [13].
- The database tier -- This software stores application data.

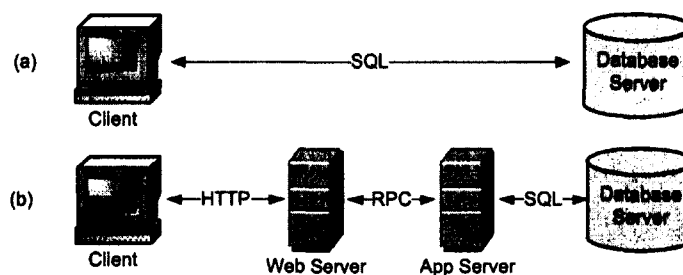


Figure 2.1: The Client/Server and n-Tier Model

2.1.2 Admission Control in Web Servers

Admission control is mostly focused on web servers to prevent computing system from being overloaded. The Apache server binary is called *httpd* in Linux and runs

as daemon processes that listen to the specified socket port. When a request appears, the server attaches a child process to it by either spawning a new one or awakening one from process pools. The request is then passed to that child process for processing. *MaxClients* sets the limit on the number of simultaneous requests (i.e. worker pool) that will be served, thus imposing a limitation on the processing capacity of the server. A large *MaxClients* value may allow Apache to handle more client http requests. However, this high value can also result in excessive resource usage that finally increases the response time dramatically.

2.1.3 Related Work on Admission Control

There has been a lot of research in the areas of overload control, service differentiation, request scheduling, and request distribution for web servers and web server clusters. We provide an overview here.

The use of admission control for overload management has been proposed and explored in several systems. Iyer et al. [78] employ a simple admission control mechanism based on bounding the length of the web server listen queue. The authors try to minimize the work spent on a request which is eventually not serviced due to overload. They analyze different queue management approaches and use multiple thresholds, though they do not specify how these thresholds should be set to meet a given performance target. Cherkasova and Phaal [54] introduce session-based admission control, driven by a CPU utilization threshold, which performs an admission decision based on user sessions rather than individual requests. During periods of overload, it rejects new sessions while serving requests from already accepted sessions. Carlstrom and Rom [42] proposed a performance model for scheduling client requests and session-level admission control using a generalized processor scheduling discipline. To improve the efficiency of session-based admission-control mechanisms and reduce its overhead, Voigt et al. [121, 122]

present several kernel-level mechanisms for overload protection and service differentiation. In general, these earlier works consider a single tier web server and the proposed techniques do not directly provide a solution for a multi-tier system.

Many of the proposed techniques are based on static admission policies, such as bounding the maximum request rate of requests to some constant value. For example, PACERS [51] limits the number of admitted requests based on estimated web server capacity. The authors use a very simple simulated service where request processing time is a linear function of the requested web page size. Similar ideas (and similar problems with fixed threshold settings) are pursued in [113]. Web2K presents a mechanism prioritizing requests into two classes: premium and basic. Connection requests are forwarded into two different request queues. Admission control is performed using two metrics: the accepted queue length and measurement-based predictions of arrival and service rates from that class. Bartolini et al., in their recent work [35, 36], introduce a quite elaborate session admission algorithm, called AACA, that self-configures a dynamic constraint on the rate of incoming new sessions to satisfy Service Level Objectives (SLO) guarantees. The rate limitation for the next iteration interval is based on a relatively straightforward prediction of the session arrival rate from the previous interval measurements.

Many early papers combine differentiated services with admission control [30, 60, 80, 86, 122]. Kanodia and Knightly [80] develop an admission control and service differentiation mechanism which is based on a general framework of request and service envelopes. Such envelopes statistically describe the server's request load and service capacity as a function of the interval length. The proposed mechanism integrates latency targets with admission control and improves the percentage of requests that meet their Quality of Service (QoS) delay requirements. The approach is evaluated via a trace-driven simulation. A number of systems have explored a controlled content adaptation [28, 46, 62] for scaling web site performance, i.e., de-

grading the quality of static web content by reducing the resolution and the number of images delivered to clients. This helps to reduce the use of server memory and network bandwidth.

Several research papers have examined how control theory can be applied in the context of web servers [29, 89, 103]. Lu et al. [89] present a control-theoretic approach to provide guaranteed relative delays between different service classes. The main challenge in such works is that good models of system behavior are difficult to derive. Web applications are subject to widely varying traffic patterns and resource demands. Linear models may be inaccurate in describing systems with bursty loads and resource requirements. Lama et al. [84] combine neural fuzzy control theory and machine learning techniques for performance assurance. The parameters and structure of the neural fuzzy controller are dynamically “learned” at run time. The structure learning phase dynamically determines the input node space and fuzzy logic rule nodes depending on the measured error and change in errors. The parameter learning phase adaptively modify the position and shape of membership functions to mitigate dynamic workload variation. Urgaonkar et al. [117] argue that dynamic resource provisioning of multi-tier applications is very different from provisioning of single tier applications. The authors design an analytical model of multi-tier applications that practically reflects the required capacity at different tiers for a given workload. The authors employ a combination of predictive models and reactive techniques at different time scales for dynamic resource provisioning.

Many earlier papers study the additional request and connection scheduling for improving web server performance [52, 57, 61]. While shortest job first scheduling for static content web sites can improve performance of a web server, it can not prevent it from overload. Elnikety et al. [61] present an elegant solution for admission control and request scheduling for multi-tier e-commerce sites. Their method is

based on measuring the execution costs of online requests, distinguishing different request types, and performing both overload protection and preferential scheduling using a straightforward control mechanism. They implement their admission control using a proxy, called Gatekeeper, with standard software components on the Linux operating system. There exists a few other works close to Gatekeeper in spirit; SEDA [125] is a prime example. In SEDA, applications consist of a network of event-driven stages connected by explicit queues. SEDA makes use of a set of dynamic resource controllers by preventing resources from being over-committed when demand exceeds service capacity. It keeps stages within their operating regime despite large fluctuations in load and allows services to be well-conditioned to load, i.e., preventing their performance degradation under severe overload. The authors describe several control mechanisms for automatic tuning and load conditioning, including thread pool sizing, event batching, and adaptive load shedding.

2.2 Virtualization of Data Centers

2.2.1 Background

Virtualization is not a new technology, it was first developed during late 1960s and early 1970s. In a virtualized system environment, a hypervisor or *Virtual Machine Manager* (VMM) is a layer of software that manages the allocation of hardware resources, and also creates, and runs virtual machines. The real hardware resources are owned by the VMM and it is its responsibility to make the resources available to one or more guest operating system that alternately execute on the same hardware. Thus, a *guest* operating system is given the illusion of owning a complete set of standard hardware.

The first version of IBM virtual machine operating system was VM/370 (or officially Virtual Machine Facility/370) released in 1972 [56]. VM/370 was built as a

general purpose OS for IBM System/370 mainframe machines. The virtualization features are mainly used for supporting time-sharing systems, maintaining backward compatibility of IBM System/360, and providing a private, secure and reliable computing environment [56]. The virtual machine manager of VM/370 was called the *control program* (CP). It ran on the physical hardware to create the virtual machine environment. Virtual machines ran a single-user, lightweight operating system called the *conversational monitor system* (CMS). The CP/CMS design successfully makes a separation of resource management and of the services that users cared about. With the rising of personal computers, interest in these classic virtualization techniques faded.

Virtualization has regained its popularity in recent years because of the promise of improved resource utilization through server consolidation, guaranteed resource allocation, and performance isolation. Disco [41], one of the first research operating systems, has led to a wide range of commercial virtualization techniques [9, 10, 21, 34].

2.2.2 Virtualization Conditions

In a classic paper [105], Popek and Goldberg formulate the sufficient conditions for an instruction set architecture (ISA) to efficiently support virtual machines. According to Popek and Goldberg, there are three properties that a VMM must satisfy: *efficiency*, *resource control*, and *equivalence*.

1. *Efficiency* means that a statistically dominant subset of machine instructions must be executed directly by the real processor, with no software intervention by the VMM.
2. *Resource control* means the VMM must have complete control of the virtualized resources.

3. *Equivalence* means that any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with only a few exceptions.

Popek and Goldberg [105] derive the sufficient (but not necessary) conditions for virtualization in a famous theorem. They first divide an ISA into three different groups:

- **Privileged instructions:** Those that trap if the processor is in user mode and do not trap if it is in system mode.
- **Control sensitive instructions:** Those that attempt to change the configuration of resources in the system.
- **Behavior sensitive instructions:** Those whose behavior or result depends on the configuration of resources (the content of the relocation register or the processor's mode).

With the above definition, Popek and Goldberg state that:

Theorem 1. *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

Their reference to "third generation computer" is an integrated circuits based computer with a processor and linear, uniformly addressable memory. The assumptions regarding the operation of "third generation computer" are: relocation mechanisms, supervisor/user mode, and trap mechanisms [105].

The theorem says that if sensitive instructions executed in the user mode always trap to the VMM (force control to go back to VMM), an efficient virtual machine implementation can be constructed. All the non-privileged instructions can be executed natively on the host platform and no emulation is needed.

It is known that Intel x86 ISA has several instructions that are sensitive but not privileged [106]. They do not trap when they are executed in user mode. Therefore Intel x86 ISA violates Theorem 1. However, since the theorem provides a sufficient but not necessary condition, it does not mean that it is not virtualizable. It means that additional steps must be taken in order to implement a virtual machine with possible loss of some efficiency. For convenience, the instructions that are sensitive but not privileged are called *critical* instructions. The VMM can scan the guest code when it is first executed and replace the critical instructions with a trap to the VMM. This process is known as *patching* [112].

Paravirtualization is another technique used to support high performance virtual machines on x86 hardware. Paravirtualization presents a software interface to virtual machine to a system that is similar but not identical to the underlying native hardware and requires making modifications to the guest operating system [126]. Xen [34] is an example system that specifically targets the Intel IA-32 ISA. As mentioned, Intel x86 ISA has critical instructions that are difficult to be efficiently virtualized. The Xen [34] system takes the hosted operating system, such as Linux or Windows, and makes minimal modifications to the machine-dependent parts of the system to eliminate the need to perform complex virtualization tasks such as patching.

In this thesis, we focus on the Xen and VMware [23] virtualization platforms. Both systems support fine grain management of memory and CPU resources, as well as the ability to transparently migrate running virtual machines from one physical server to another.

2.3 Server Consolidation

One of the key applications of using virtualization in data center is server consolidation. The idea is to take under utilized servers in the corporations, convert them into VMs and run them on a smaller number of physical servers, thus achieving a better utilization of hardware resources. This dissertation first introduces several fundamental virtualization techniques and present the server consolidation related work.

2.3.1 Virtual Machine Migration

Virtual machine migration refers to transfer the entire virtual machine -- the in-memory state of the kernel, all processes, and all application states across distinct physical hosts. Migration can be either *live* or *cold*, with the distinction based on whether the instance is running at the time of migration. In a cold migration [115], the virtual machine is powered off, saved and sent to another physical machine. In a live migration [55], the domain continues to run during transfer and downtime is kept to a minimum.

Xen live migration requires multiple stages [55]. It begins by issuing a request from host A (source), to host B (target), reserving the resources that the source will need. If the target acknowledges the request, the source moves into the *iterative pre-copy* stage, in which the source copies all memory pages to the target through a TCP connection. While transferring, memory pages in the source could be changed or marked dirty and these pages are copied in the next round of transfer. Xen iterates the memory transfer until only a set of very frequently changed pages is left and begins the *stop-and-copy* stage. Host A suspends the running OS and copies the remaining pages to host B. During the final *commitment* stage, the target informs the source that a complete OS image is received and reinstantiates the

migrated VM.

2.3.2 Performance Isolation Among Virtual Machines

During server consolidation, multiple under-utilized virtual machines are packed into a single physical host, sharing the available hardware resources including CPU(s), memory, network adapter(s), and disk(s). This causes unpredictability in the performance of each individual VM. In this occasion, it is desirable to provide mechanisms that can prevent VMs from monopolizing resources and guarantee predictable performance. Multiple techniques are used to multiplex physical hardware across VMs.

For the CPU resource, multiple scheduling techniques are proposed in order to guarantee that every running VM receives some amount of CPU time. For example, on the Xen hypervisor, the *Borrowed Virtual Time* (BVT) [59], *Simple Earliest Deadline First* (S-EDF) [4], and the *Credit scheduler* [3] have been used for controlling how the computing power is distributed among competing VMs. On recent versions, Xen uses the credit scheduler as the default choice [115]. This scheduler provides two properties for each domain: a *weight* and a *cap*. The weight is a relative value, e.g., a domain with a weight of 512 gets twice as much CPU as a domain with a weight of 256 on a contended host. In contrast, the cap is an absolute value, expressed in percentage of one physical CPU. A comparison study of these three schedulers have been conducted by Cherkasova et. al. [53].

Regarding memory, *memory ballooning* is a technology for a virtual machine to give up memory or to request more memory from the hypervisor. It was first introduced in VMware ESX [123]. VMM controls a *balloon* module running in the virtual machine. When VMM wants to reclaim memory, it instructs the driver to "inflate" the balloon. Inflating the balloon calls the kernel interface in virtual machine to allocate physical memory and return these pages to VMM. Similarly, VMM may

deallocate the memory by instructing it to ``deflate" the balloon.

Regarding storage, the local I/O bandwidth management at each host was done using Start-time Fair Queuing (SFQ) [79]. However, in enterprise applications, host level scheduling is not sufficient, since multiple hosts can access the same storage array. There are prior works that provide mechanisms for allocating storage resources to individual VMs [68, 70]. *mclock* [70] is an IO scheduler that provides resource controls (shares, limits, reservations) for storage array at a per-VM level. This is known as Storage I/O control and released in VMware's vSphere5 [119].

For the network resource, it is possible to use network traffic shaping techniques to enforce limit and weight-based allocation [22, 39]. In VMware's vSphere, network I/O controls are implemented by three key software layers: teaming policy layer, shaper, and scheduler. Since virtual machines and physical machines could be configured with multiple Network Interface Controllers (NICs), team policy layer is to determine which traffic from virtual ports will be sent over which physical NICs. Shaper layer enforces the configured *limit* parameter. For example, if one VM's traffic is limited to 1Gib, any additional traffic is dropped by the shaper, even if physical NIC has the capacity. Finally, one scheduler is instantiated for each physical NIC and it distributes the network bandwidth among VMs based on their *shares* value.

2.3.3 Related Work on Consolidation

Consolidating multiple applications on a single physical server can solve issues related to low utilization, however, how to autonomically and accurately perform server consolidation at enterprise level is still an unsolved research problem that faces significant technical challenges [120] including how to accurately measure and characterize an application's resource requirements, how optimally to distribute the virtual machines hosting the applications over the physical resources, how much resource each virtual machine should be allocated, and how to balance the

workloads at run time when applications and servers become overloaded.

To solve these questions, there has been early studies in the areas of application measurement and characterization, server consolidation strategies, and dynamic resource management. We provide an overview here.

2.3.3.1 Measurement and Characterization

To support autonomic application management functions, we need an accurate monitoring infrastructure reporting resource usage of different VMs. However, the standard monitoring systems which directly profile VM resource utilization inside the VM might not reflect the true usage of resources by different VMs. The reason is that virtualization of I/O devices or network devices results in a model where the data transfer process involves additional system components, e.g., hypervisor or device driver domain. Hence the resource usage when the hypervisor or device driver domain handles the I/O or network data on behalf of a particular VMs needs to be charged to the corresponding VM. Meanwhile, disk I/O activities measured at VM and hypervisor level could have significant differences due to page cache or write coalescing mechanism in VMM. In sum, real application resource consumption in virtualized environment can be quite different from its measured usage because of additional virtualization overhead and interactions with the underlying VMM.

Several early papers measure the impact of virtualization overhead on benchmarks. Gupta et al. [72] present the design and evaluation of a set of primitives implemented in Xen to enforcing performance isolation among VMs. They look into per-VM CPU overhead in the driver domain caused by network traffic and use a linear model to approximate their relationships. In this thesis, our work is complementary to [72] on driver domain CPU overhead modeling and extends to other resources including disk I/O and memory. Disk I/O activity is also related to the CPU overhead. Wood et al. [127] investigates the virtualization overheads and use it to

accurately predict the resource needs of virtualized applications, allowing them to be smoothly transitioned into a data center. They propose a combination of application modeling and virtualization overhead profiling for estimating the hypervisor and virtual machine CPU utilization of an application. They use micro-benchmarks to profile the relationships of different I/O activities to the CPU overhead, apply robust stepwise linear regression method to build the models, and predict an application's CPU demand after virtualization based on the benchmark models and the application's native resource utilization. The work presented in this dissertation is different from theirs in the following aspects: (1) our calibration process is a run-time process where a feedback loop controls the remodeling process, while their prediction process is a one-time offline profiling with a fixed set of regression models; (2) our calibration process covers three other resources in addition to CPU, and there are situations where the virtual activities are not equal to their physical activities for some non-CPU resources. (3) our DFG method is a source separation framework where different functional modeling approaches can be used as plug-ins, as it is not limited to linear regression.

Isaci et al. [76] study the run-time CPU demand estimation in VM consolidation for effective dynamic resource management. They derive a simple and accurate alternative estimate of CPU demand even when a server is overloaded with VMs hosting CPU-intensive applications. Extending their idea to other type of applications (e.g., IO-intensive) and other type of resources is interesting and important. Pacifici et al. [101] consider a dynamic CPU demand estimation problem for web applications. They use statistical and classification methods to determine the CPU demand for different web request types.

Since many virtualization platforms introduce additional virtualization overhead, many research works [65, 107, 110, 118] provide a capability to scale the resource usage of the original workloads by a specified multiplier. For some applications it

might be a reasonable approach, however, in general, additional CPU overhead highly depends on system activities and operations performed by the application. Simplistic constant scaling may result in significant modeling error and resource over-provisioning.

Virtualization technologies evolve in a fast speed, and many new approaches have been proposed to address virtualization overhead concern. For example, Liu et al. [87] propose hypervisor-bypassing in Xen to reduce the performance penalty of network I/O; Santos et al. [109] designs an optimized network IO scheduling algorithm to improve network throughput in Xen. These results bring more dynamics into the relationships between physical and virtual resource activities, and call for the necessity of an adaptive calibration solution like the one presented in this thesis.

2.3.3.2 Consolidation Strategies

Virtual infrastructure platforms typically include software that can help to balance virtual machine workloads across hosts and to locate VMs on the best possible servers for their workload in a resource pool; VMware Distributed Resource Scheduler (DRS) [75] and XenServer Workload Balancing [27] are examples of load balancing solutions. However, both of these solutions require to manually tune the *weightings*. The weightings are a way of ranking resources according to how much you want them to be considered and are used to determine the processing order. That is, after workload balancing determines its needs to make a recommendation, it uses specifications on the importance of resources to determine which host's performance to address first and which virtual machines to recommend migrating first. As this dissertation illustrates in Section 6.2, it is very hard to find a general approach for utilization normalization across different application under different workloads.

Wood et. al [128] present Sandpiper, a system that automates the task of mon-

itoring, detecting and migrating hotspots to least loaded server. They define a new metric *volume* as the product of server's CPU, network and memory loads. The *volume* captures the degree of (over)load along multiple dimensions in a unified fashion and can be used by the mitigation algorithms to handle all resource hotspots in an identical manner. Sandpiper is designed to balance the volume across all physical servers. They implicitly assume that each resource has the same weight when balancing.

Bejerano et al. [38] study the user-AP associations for max-min fair bandwidth allocation in wireless LANs. They showed the strong correlation between fairness and load balancing, and devised load balancing algorithms that achieve constant-factor approximations. Their work extended the long-lined networking research on fair bandwidth allocation [66, 82].

Ghodsi et. al [64] present dominant resource fairness (DRF), a methodology that generalizes max-min fairness to the field of multiple users making heterogeneous demands on multiple resource types. In the DRF model, users record their task requirement using a demand vector of two metrics, CPU and memory. It applies the max-min fairness to the user's dominant resource to balance the load. This dissertation considers four dimensional resources (CPU, memory, network, and disk) for min-max load balancing and present a multi-class closed queuing model to predict the performance of multiple applications in order to select the best performance.

Lee et. al [85] study the performance degradation problem of VM consolidation. For computation and network resources, there is no performance degradation in low utilized workload while performance degradation in the presence of high resource contention is gradual and fair. This observation agrees with the motivation in minimizing the maximum server allocated load to reduce the competition for resources and to improve application performance.

2.4 Feedback Control-based Resource Allocation

Early studies of dynamic resource allocation in distributed systems have largely focused on allocating resources across multiple physical nodes. In [47], cluster power management is done by allocating resources appropriately to maximize the global utility, while minimizing the power usage. In [111], an integrated framework is proposed by combining a cluster-level load balancer and a node-level class-aware scheduler to achieve both overall system efficiency and individual response time goals. However, these existing techniques are not directly applicable to allocating resources to applications running in VMs. They also fall short of providing a way of allocating resources to meet end-to-end application SLOs.

To meet a target SLO for a multi-tiered web-application, [40] presents a methodology that automatically determines the amount of required resources expressed as an integer number of EC2 instances of a specific type. That implies that application resources are scaled horizontally in coarse-grained VM instance increments. In contrast, in [33], resource containers are proposed to achieve fine-grained resource control for applications. In [123], new memory management techniques are proposed to allow dynamic re-allocation of memory between different VMs. The work presented in this dissertation relies on similar management techniques from modern hypervisors such as VMware ESX [20], Xen [34], and Microsoft Hyper-V [26].

Control theory has been successfully applied to the resource management of computer systems [74, 81]. In [29], a control loop is designed to guarantee Web server performance via online content adaptation. Similar techniques are used to dynamically adjust the cache sizes for multiple request classes [94]. In [131], application level resource management with feedback is achieved by having “friendly” VMs that adjust their resource demands for fair resource sharing. In [100], the au-

thors model performance interference between co-located VMs and apply closed-loop control to mitigate such interference if feasible. In this dissertation, we directly model the relationship between the application performance and the resource utilization levels of individual VMs, in effect taking into consideration implicitly any performance interference.

Many feedback control techniques manage only one type of resource. For example, AppRaise [124] is a system that uses queuing models to represent application performance in a virtualized environment and applies predictions from the models to put CPU caps on virtual machines. Multiple resources are managed in [63, 102]. In [63], the authors apply multiple-input multiple-output control to tune two configuration parameters within a single Apache Web server to regulate its CPU and memory. AppRM also manages multiple resources (CPU and memory), using online models instead of the offline models used in [63].

A similar two-level resource control architecture was presented in [129], where a local controller estimates the amount of resource needed by each VM using a fuzzy-local-based modeling and prediction approach, and a global controller runs at each host (aka. node) to mediate the resource requests from different local controllers. This work considers only one resource type (CPU) and applications hosted in a single VM. Another two-level resource control system in [102] applies online statistical learning and adaptive control theory to translate the SLO of a multi-tiered application to the capacity requirements for multiple resource types (CPU and disk I/O) in multiple VMs, which is the approach we adopt in this dissertation. We stress that our work differs from both [129] and [102] in the following aspects: (1) Both prior works use CPU *limit* and neither utilizes *reservation* for any resource type, which we believe to be a powerful resource control knob whose utility should be explored. In contrast, AppRM employs all three resource control knobs (reservation, limit, shares). (2) The higher-level controller in both papers deals with a single physical

host with fixed capacity, whereas our RP Manager deals with a logical container such as a resource pool whose capacity itself can be a moving target. The latter is a unique resource allocation model supported by VMware DRS [69]. (3) Both the global controller in [129] and the node controller in [102] handle the resource requests from individual VMs at the same time and with the same frequency, which poses synchronization constraints on the lower-level controllers. The RP Manager in AppRM interacts with multiple vApp Managers asynchronously so that each vApp Manager can work at its own pace based on the application need. As a result, AppRM provides a holistic resource management tool that works seamlessly within the hierarchy of a virtual data center, i.e., across multiple resource types, multiple applications and VMs, but more importantly within the resource pools where these VMs are located.

3 Admission Control for Busy Multi-tier Services

Capacity planning plays an important role in “sizing” IT systems and needs to be even more effective in case of e-commerce sites where customers have high expectations for QoS support, given an environment that is characterized by unpredictability. Over-provisioning offers only a partial solution as its benefits may be offset by higher energy and operating costs of a system that is rarely needed to be that large. To contain the size of the system and yet maintain user-perceived performance levels in the form of service-level objectives (SLOs), several methodologies have been proposed that rely on admission control and/or techniques for service differentiation that are threshold based [29, 35, 36, 54, 80]. Yet, we show in this chapter that these techniques may be unable to provide robust business solutions. If the site experiences temporal surges in user arrivals or service demands (i.e., bursts) [98, 99], triggered by sales or seasonal events, then threshold-based overload control is largely ineffective.

To get the intuition on why prevailing techniques may not be effective for system management under bursty conditions, consider a web service that is built according to the industry-standard, multi-tier paradigm. Typically, a user access to a web service occurs in the form of a *session* consisting of many individual requests. Placing an order through the web site involves further requests relating to selecting a

product, providing shipping information, arranging payment agreement and finally receiving a confirmation. For a customer trying to place an order, or a retailer trying to make a sale, the real measure of a web server performance is its ability to process the entire sequence of requests needed to *complete* a transaction. Utilization-based policies [29, 54] accept a new session only if there is enough capacity in the system to guarantee that future requests of this session can be processed and the entire session can complete successfully. If the system operates on or above a certain capacity threshold, then a new session is rejected (or redirected to another server, if available).

In a multi-tier system that operates under bursty workload conditions (in the form of bursty arrivals and/or bursty service demands at tiers), threshold-based policies become ineffective. The main reason is that if flows are bursty, then the system is subject to the phenomenon of *persistent bottleneck switch* [98]. When this phenomenon is present, average utilizations of the various tiers may be moderate, but during a workload burst the system may experience nearly simultaneous arrivals of requests in a tier that gets overloaded for a period of time. After the tier processes these requests, they arrive again nearly simultaneously on the next tier, which now experiences a period of overload. Interleaving time periods of intense activity with almost no activity on the various tiers results in persistent bottleneck switch, i.e., the bottleneck continuously shifts from one tier to the next across time, hindering the effectiveness of a threshold based policy. Several questions are raised, including whether it is advisable to activate a control on one tier (e.g., the bottleneck tier), multiple tiers, or all tiers, and under what conditions.

In this chapter, we design a solution to the above problem by first studying the reasons why threshold-based policies that are documented to work well in single-tiered system may fail in a multi-tiered system with bursty workloads. We show that threshold-based policies have a slow reaction to bursts, and therefore cannot

maintain low ratios of aborted sessions. The new solution that is offered here can be summarized as follows: we aim to dynamically control the number and the type of user requests admitted for processing into the multi-tier system and continuously differentiate between requests from already accepted sessions and requests for new sessions. When the system enters the overload state, we advocate buffering of requests from the already accepted sessions in a so-called “blocking” queue, that effectively acts as a waiting room [32, 104]. This blocking queue successfully differentiates among the requests of already accepted sessions to those of new sessions, and implicitly gives them a higher priority.

The performance of accepted sessions remains directly bounded by the time the accepted requests spent in the blocking queue. The larger the size of the blocking queue, the lower the number of aborted sessions but at a cost that may result in SLO violations due to additional waiting in the queue. We perform a sensitivity study to explore the different blocking queue limits under a variety of burstiness profiles. The conclusion is that the effectiveness of the proposed construction is strongly related to the workload burstiness. Based on this analysis, we present an effective blocking mechanism that autonomically adjusts the blocking queue capacity to the degree of burstiness of the workload. Note, that in this chapter, we consider the response time as the target SLO. The designed approach can be used similarly for different target SLOs (e.g., loss probabilities in terms of aborted or dropped requests).

The resulting policy is an autonomic session-based admission control policy, called *AWAIT*, that adjusts the blocking queue capacity in response to workload burstiness. We perform detailed simulations using the TPC-W benchmark with extended functionality for generating bursty session arrivals [99] to demonstrate the effectiveness and robustness of the new strategy. *AWAIT* supports a simple and inexpensive implementation. It does not require significant changes or modifications

to the existing Internet infrastructure, and at the same time, it significantly improves the performance of overloaded multi-tier web sites. Extensive experiments illustrate *AWAIT*'s ability to closely maintain target SLOs across realistic workloads (where the degree of burstiness changes over time) by effectively adapting the blocking queue size to the workload bursts.

3.1 Motivation: Capacity Planning and Admission Control

In this section, we present a short case study that illustrates how burstiness may impact the performance of admission control. We present some initial experiments that illustrate the problem and show that burstiness can spoil the effectiveness of an admission control mechanism that is deployed on a single-tier.

3.1.1 Basic Capacity Planning

Overload management is a critical business requirement for today's Internet services. A common approach to handle overload is to apply specific resource limits that typically bound the number of simultaneous socket connections or threads. For example, in traditional web servers that employ thread-per-connection implementation, the server configuration specifies the number of processes (and connections) that are allocated for admitting the user requests. Therefore, in the Apache web server [1], when all the server threads are busy, the system stops accepting new connections. The same principle applies for providing the basic overload protection in multi-tier applications. The system administrators may set limits on the number of simultaneous client sessions (we call them *active requests*) in the system. Limiting the number of active requests is critical for quality of service: setting this limit too low results in achieving a good response time but at a price of lower system

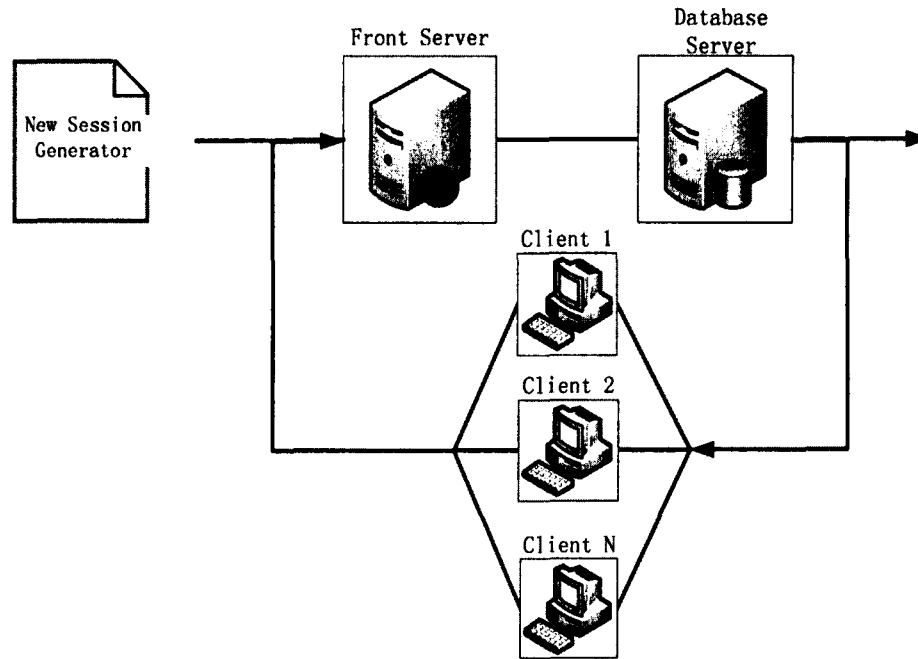


Figure 3.1: The basic structure of TPC-W model

throughput and also of high number of dropped user sessions. Setting this limit too high may lead to better throughput and reduced drop rates but at a price of a much higher user response time.

Figure 3.1 shows a high-level model of an e-commerce site used in this chapter. It is based on the TPC-W benchmark implemented as a multi-tier application. It consists of a web server, an application server, and a back-end database. The web server and the application server usually reside within the same physical server, which is called a front server. After a new session connection is generated, client requests circulate among the front and database server before they are sent back to the client. After a request is sent back, the client spends an average think time $E[Z]$ before sending the next request. A session completes after the client has generated a series of requests. The TPC-W benchmark defines 14 transactions, that can be generally classified as ``browsing" or ``ordering". There are three widely used transaction mixes: browsing, shopping, and ordering.

We first focus on the *capacity planning* aspect and design an experiment that

can first identify the number of simultaneously active requests that can guarantee a certain QoS to the end user. We use the benchmark's ordering mix, that consists of 50% browsing and 50% ordering transactions. Request service times in the front and database servers are derived using the methodologies and models presented in [43, 98] that have been shown to capture very accurately the performance and behavior of TPC-W. Consistent with the specifications of the TPC-W benchmark, the average user think time is equal to 7 seconds, exponentially distributed, i.e., here is no burstiness in the arrival stream of new sessions. We set new session arrivals with a constant rate of 35 requests per second. Each session consists of a sequence of requests (i.e., essentially a series of visit "rounds" to the front and database server that define the session length) that is uniformly distributed with parameters 5 and 35, that is with expected mean equal to 20.¹ Note that the mixture of requests for new and existing sessions is not pre-defined but determined by the average user think time and the session length. In this experiment, the ratio of new to existing session is close to 0.2.

It is a typical situation when after a certain waiting time an impatient client might "click again" and reissue the original request. Client request timeouts and retries can be added to our model to reflect a more complex and realistic scenario. A client with a timeout value of t sec can be considered as an additional QoS requirement: A request latency must have a limit of t sec. If this requirement is not met, after a given number of retries, the session is aborted. This could decrease the useful system throughput (due to the processing overhead of these additional requests) but it would not fundamentally change the results of our study [54]. In this chapter, we use a simplified model without request timeouts and retries in order to focus on the effects of burstiness.

¹We could have used another distribution or different parameters to the uniform distribution of visit rounds. Experiments with different parameters are qualitatively the same as the results presented here and are omitted due to lack of space. Modeling tier visits with a uniform distribution is consistent with experimental results in [43, 98].

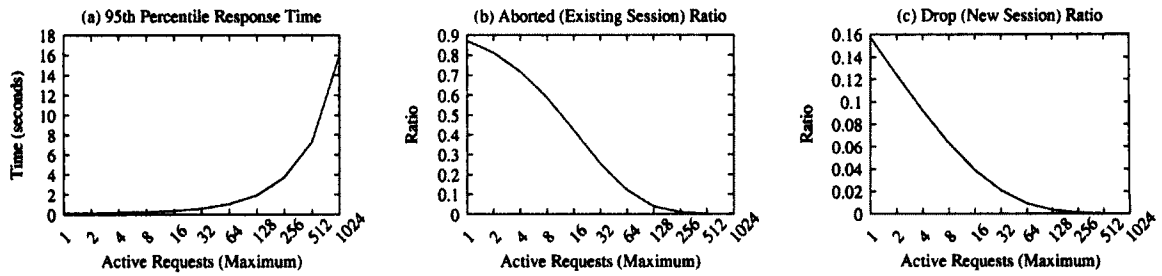


Figure 3.2: Capacity Planning study for SBAC under exponential (i.e., not bursty) new session arrivals. Performance measures are presented as a function of the maximum number of active requests in the system.

Figure 3.2(a) illustrates the 95th percentile of user end-to-end response time as a function of the predefined value of the maximum active requests in the system. We define the *ratio of aborted sessions* as the ratio of aborted accepted sessions to the total number of accepted sessions. The *new session drop rate* is defined as the ratio of dropped new sessions over the generated sessions. Figures 3.2(b) and 3.2(c) present the aborted rate of accepted (existing) sessions and the drop rate of new sessions, respectively, as a function of the allowed active requests. If an SLO of 4.0 seconds is the performance objective for request's response time, then Figure 3.2 suggests that one may use 256 as the recommended limit on active requests. This value strikes a good balance among all desired measures: the request response time (below 4 seconds) and the minimized number of aborted and dropped sessions. Note that the value of 256 is a configuration parameter of web server set by the system administrator and by no means a parameter of the *AWAIT* controller that we introduce in this chapter.

Table 3.1 summarizes the system configuration parameters that are used in the remaining of this chapter.

3.1.2 Introducing Burstiness

Because our purpose is to evaluate the different admission control algorithms under bursty arrival conditions, we introduce here three burstiness profiles that we use in

Table 3.1: Configuration

| | |
|--------------------------|--|
| average user think time | 7 sec. |
| new session arrival rate | 225 req/min. |
| session length | uniformly distributed between 5 and 35 |
| active request limit | 256 reqs |

the rest of this chapter. We use a Markovian Arrival Process (MAP) to generate three arrival processes. For details on the generation of the three MAP processes as well as on their effectiveness in mimicking bursty arrivals such those reported in the 1998 World Cup web server we direct the reader to [99]. MAPs have also been shown to be surprisingly compact yet very effective models of the service process in multi-tier systems, modeling implicitly conditions such as caching or database locks (see [98]).

The burstiness profiles (i.e., the number of arrivals as a function of time) for the three MAPs that we use for the arrival process are illustrated in Figure 3.3. The three levels are distinguished by the way the number of active clients fluctuates across time, with burst level 1 showing moderate fluctuation, while burst level 3 showing periods of intense activity to alternate with periods of negligible activity. We stress that the distributions that correspond to the three processes share the same mean and coefficient of variation. In the appendix, we provide the configuration parameters used for these three MAPs, as well as pseudo-code for reading the MAP configuration file and for generating MAP random variates.

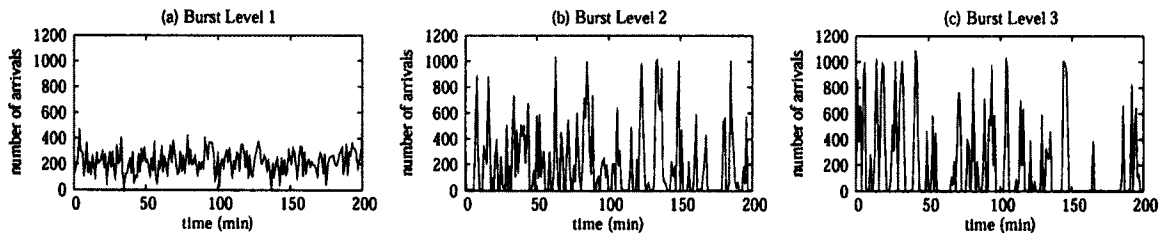


Figure 3.3: The burstiness profiles of the three arrival MAPs.

3.1.3 Burstiness in Flows and Admission Control

Here we investigate the effect of burstiness in the arrival process to a classic admission control algorithm. Session-based admission control (SBAC) [54] has been shown to be the effective policy for web servers. It is based on monitoring the CPU utilization of the web server. SBAC accepts a new session only when the system utilization is below a certain threshold, to guarantee a successful session completion. If the observed utilization is above a specified threshold, then for the next time interval, the admission controller rejects all new sessions and only serves requests from already admitted sessions. Once the observed utilization drops below the given threshold, the admission controller changes its policy for the next time interval and begins admitting and processing new sessions again. A web server employs a configurable size queue for buffering the incoming requests. If the arriving request belongs to the already accepted session and the queue is full, then the entire session is aborted. The *useful throughput* of the system is measured as a function of the number of completed sessions. Aborted requests of already accepted sessions are highly undesirable because they compromise the server's ability to process all requests needed to complete a transaction and result in wasted system resources.

We have implemented the SBAC mechanism in a simulation model of a client-server system that is built according to the TPC-W specifications. The SBAC mechanism uses a front server utilization threshold for admitting new sessions.² Figure 3.4 illustrates the ineffectiveness of the threshold-based techniques in the presence of bursty arrivals. We compare the results of two different admission control strategies. A first strategy (called *baseAC*) employs a traditional overload control

²For the TPC-W testbed with the ordering mix, SBAC is based on the CPU utilization of the front server because the front server is the system bottleneck for this particular mix. In general, the admission control should be based on the utilization of the bottleneck resource, e.g., if the DB tier is a bottleneck then its CPU utilization should be used for SBAC decisions.

based on admitting a fixed, predefined number of active requests for processing. Here, we set *ActiveRequests* = 256 as suggested by capacity planning (see Figure 1). The second strategy is SBAC where the front server utilization threshold is set to 85% and 95% respectively.

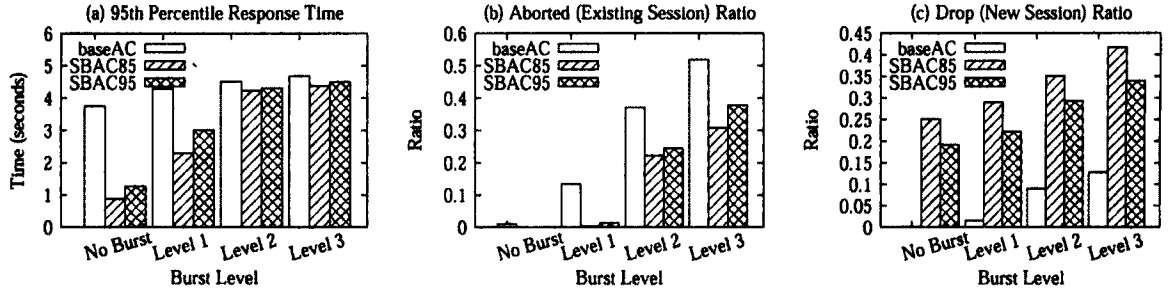


Figure 3.4: Three different burstiness profiles. The capacity planning results and SLO targets are now violated. It appears that a queue size of 256 (i.e., maximum active requests for the baseAC configuration) is not sufficient to meet SLO requirements.

Figure 3.4(a) illustrates the 95th percentile of user response time. While SBAC is effective in maintaining good response times under bursty arrivals, this is achieved at the expense of a relatively high ratio of aborted sessions as well as a high ratio of rejected new sessions, see Figure 3.4(b)-(c). The *baseAC* strategy does not differentiate between the requests from new and existing sessions and this leads to a very high ratio of aborted sessions.

Both of these threshold-based strategies might be a reasonable choice under non-bursty traffic. However, they clearly exhibit their deficiencies under bursty traffic conditions. This simple experiment shows that the admission control mechanism has to take traffic burstiness into account and adapt the system configuration and/or thresholds in order to effectively deal with bursty traffic conditions. In the next section, we present a new algorithm that effectively deals with the above problem.

3.2 AWAIT Algorithm

In this section, we describe *AWAIT*, a novel session-based admission control algorithm that aims to provide additional support for bursty session arrivals. *AWAIT* has two different mechanisms to regulate request acceptance for processing. The first mechanism uses a counter of *ActiveRequests* that is defined according to capacity planning for achieving a given SLO for response time. Until this counter reaches its maximum any incoming request is accepted, this request may represent a new session or it may belong to an already accepted session. The second mechanism uses a special queue, called *blocking queue*, which is created to store the requests from already accepted sessions after the number of *ActiveRequests* reaches its maximum capacity. Figure 3.5 shows how the two mechanisms are incorporated in the TPC-W model. The *AWAIT* controller rejects new session requests if *ActiveRequests* reached its capacity but the system still admits requests from earlier accepted sessions. When the blocking queue becomes full, incoming requests from accepted sessions are unavoidably aborted. We aim to minimize the likelihood of this event, because it leads to business loss.

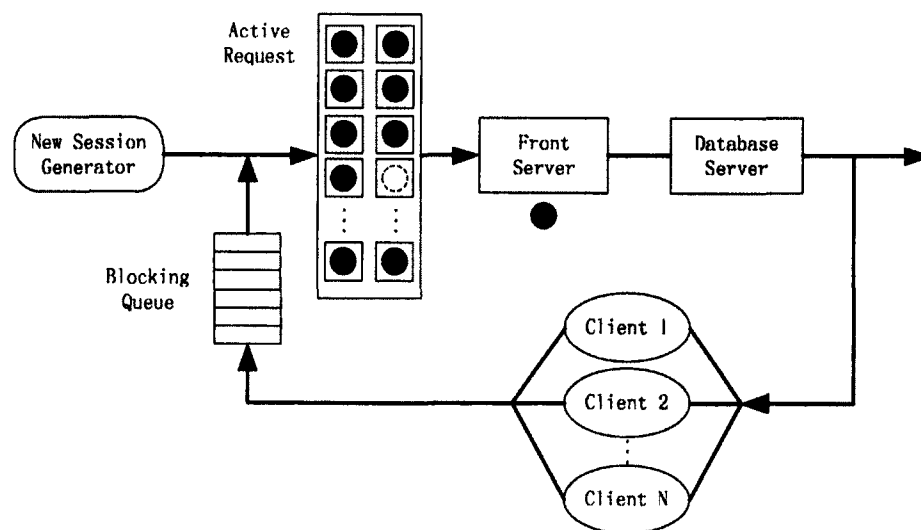


Figure 3.5: The model of AWAIT algorithm

The capacity of the blocking queue is a critical parameter for the performance of the accepted sessions since the time spent there contributes to the user end-to-end time, thus may violate the target SLOs. The larger the capacity is of the blocking queue, the longer the contribution of the time waiting there to the user end-to-end time. Similarly, the larger the capacity of the blocking queue, the smaller the expected aborted ratio of accepted requests. Striking a good balance between these two conflicting measures is the purpose of *AWAIT*.

To ease the presentation of *AWAIT*, we first present a static version that considers a fixed blocking queue size. In the adaptive version of *AWAIT*, the size of this blocking queue is autonomically adjusted according to the burstiness of the workload. In all cases, *AWAIT* ensures that the response time SLOs are met.

3.2.1 Static *AWAIT*

To formally describe the *AWAIT* algorithm, we introduce the following notions:

- *New session request* -- a request that is generated by a new client (i.e., it is a first request in a new session);
- *Accepted session request* -- a request that is issued by a client within an already accepted session;
- *ActiveRequests* -- a counter that reflects the number of accepted requests which are currently in processing by the system. These active requests could be either of new sessions or of already accepted sessions. The maximum value for this counter is set to a value defined by capacity planning (see Section 3.1). Let us denote this value as A ;
- *BlockedRequests* -- a counter that reflects the number of blocked requests which are received from the clients of already accepted sessions and which

are stored in the *BlockingQueue*. Note this difference: the blocking queue stores requests from already accepted sessions only. Let B denote the maximum value of this counter that also defines the capacity of this queue;

- *AdmitNew* -- a boolean variable that defines whether a new session can be accepted by the system. If $AdmitNew = 1$ then a new session can be accepted by the system. If $AdmitNew = 0$ then all the new sessions are rejected by the system;

Now, we describe the iteration steps of the algorithm. Let a new request *req* arrive for processing. The system can be in one of the following states.

- $AdmitNew = 1$ and $ActiveRequests < A$.

This state corresponds to normal system processing when there is enough system capacity for processing requests from new sessions *and* requests from already accepted sessions. Therefore, independent on the request type, *req* is accepted for processing and the counter *ActiveRequests* increases by one.

When this counter reaches its maximum value A , then $AdmitNew = 0$, and this corresponds to a new system state when any requests from new sessions are rejected.

- $AdmitNew = 0$ and $BlockedRequests < B$.

In this state the incoming requests are treated differently depending on their type. If the incoming request is from a new session, then it is rejected. If it belongs to an already accepted session, then it is stored in the *BlockingQueue* and the queue's counter is updated.

- $AdmitNew = 0$ and $BlockedRequests = B$.

This state reflects to the situation when *BlockedRequests* has reached its maximum value B . Any incoming request, independent on its type, is rejected. If

the request comes from an already accepted session, then its entire session is *aborted*.

Now, we describe how the system counters *ActiveRequests* and *BlockedRequests* are updated when a processed request leaves the system, i.e., the reply is sent to the client. The system can be in one of the following states (similar to the states described above).

- If $ActiveRequests < A$,
then $ActiveRequests \leftarrow ActiveRequests - 1$.
- If $AdmitNew = 0$, $ActiveRequests = A$, and $BlockedRequests = 0$,
then $ActiveRequests \leftarrow ActiveRequests - 1$ and $AdmitNew = 1$, i.e., the admission control status changes and the system again starts accepting both types of requests: from new sessions and already accepted sessions.
- If $AdmitNew = 0$, $ActiveRequests = A$, and $0 < BlockedRequests \leq B$,
then one of the blocked requests is accepted for processing in the system and only the counter *BlockedRequests* is updated: $BlockedRequests \leftarrow BlockedRequests - 1$.

We call this version of algorithm the *conservative AWAIT*. Under this algorithm the differentiation of requests from new and accepted sessions starts when *ActiveRequests* reaches its maximum value A . Then new sessions are rejected and requests from accepted sessions have extra buffering space in the blocking queue. Once *ActiveRequests* counter gets below A , then the admission restriction is lifted and new session requests are again accepted.

We also introduce a different version of the algorithm, called *aggressive AWAIT*, which at a first glance is only slightly different from the *conservative AWAIT* above. However, the performance evaluation of these two versions shows a surprising

difference in behavior and in the numbers of aborted and rejected sessions. As we see later, the *aggressive AWAIT* decreases forcefully the number of aborted sessions while supporting the same useful system throughput as the *conservative AWAIT*.

For the *aggressive AWAIT* strategy we introduce the additional variable *Overload*:

- *Overload* is a boolean variable that defines whether the system is under severe overload. Typically, $Overload = 0$ while the system can process all the requests from the already accepted sessions. $Overload = 1$ when system observes an aborted request from the accepted session. This may happen when $ActiveRequests = A$ and $BlockedRequests = B$, and the incoming request is from an accepted session. The aborted session triggers an "emergency situation" that is treated aggressively. New session requests are not accepted during overload until both blocking queue and the *ActiveRequests* in the system are flushed. This helps in providing a prolonged preferential treatment of requests from the accepted sessions to rapidly overcome the overload state.

When the overload condition is triggered, i.e., $Overload = 1$, there are slightly different rules for updating the system state when a processed request leaves the system:

- If $AdmitNew = 0$, $Overload = 1$, $ActiveRequests = A$, and $BlockedRequests = 0$, then $ActiveRequests \leftarrow ActiveRequests - 1$, but the system is considered to be still under severe overload and its admission control status does not change: the system still rejects requests from new sessions and only processes requests from the already accepted sessions.
- If $Overload = 1$ and $ActiveRequests = 0$, then the operation of the system goes back to normal: $Overload = 0$ and $AdmitNew = 1$.

The pseudo-code shown in Algorithm 3.1 summarizes both versions of the *AWAIT* algorithm: conservative and aggressive. To unify the description, in the conservative version of the algorithm the state of variable *Overload* does not change, i.e., $Overload = 0$.

Algorithm 3.1: Awaiting Admission control algorithm, aggressive version. The conservative Awaiting is obtained by removing the statements labeled Aggr.

```

for every request req that arrives for processing do
    if AdmitNew and ActiveRequests < A then
        Accept req;
        ActiveRequests ← ActiveRequests + 1;
        if ActiveRequests == A then AdmitNew ← 0 ;
    else if !AdmitNew and BlockedRequests < B then
        if type(req) == NewSession then reject req;
        if type(req) == AcceptedSession then
            accept req into BlockingQueue;
            BlockedRequests ← BlockedRequests + 1;
    else if !AdmitNew and BlockedRequests == B then
        reject req;
        // Aggressive version: trigger overload state
        if type(req)==AcceptedSession then Overload←1 ;
    Aggr

for every request req that leaves the system do
    if ActiveRequests < A then ActiveRequests ← ActiveRequests -1 ;
    if ActiveRequests==A and 0<BlockedRequests≤B then
        move one request from blocking queue to queue;
        BlockedRequests ← BlockedRequests -1;
    else if ActiveRequests == A and BlockedRequests == 0 then
        ActiveRequests ← ActiveRequests -1;
        // Aggressive version: queue flashed
        if ActiveRequests==0 then
            Overload ← 0;
            AdmitNew ← 1;
    Aggr
    Aggr

```

In sum, the rationale for the *conservative* versus the *aggressive* version of the algorithm is the following. If the system operates under a burst, then queues tend to build up fast. An accepted session that is aborted signals the system about insufficient resource capacity for processing requests from already accepted sessions. To

mitigate the performance effects of this, it is more effective to completely dedicate system resources for processing only the accepted session requests by flushing the system queues at the expense of a higher ratio of rejected new sessions. This strategy benefits accepted sessions by implicitly giving them high priority and "reserving" the system for exclusive processing of accepted session requests, until overload subsides. In the following subsection, we present experimental evidence that shows the relative performance of the *conservative* versus the *aggressive* version of the algorithm.

3.2.2 Performance Evaluation: Conservative or Aggressive?

We evaluate the performance of *AWAIT* via trace driven simulation. A high level system description of the simulated system is given in Figure 3.5. We use the same three MAPs for the arrival process as those introduced in Section 3.1. The service processes at the front server and the database server are also modeled via MAPs (see [43, 98]) that accurately capture the service demands of TPC-W's ordering mix³.

Figure 3.6 illustrates the performance of the two versions of *AWAIT* as a function of the capacity of the blocking queue B . For reference, we also report on the performance of the system with simple admission control based on the number of *ActiveRequests* only (labeled: "baseAC") as well as the performance of SBAC with CPU utilization threshold set to 85%. Note that for *all* experiments, we set the *ActiveRequests* counter to 256, as suggested by the capacity planning study of Section 3.1. The aborted existing session ratio in Figure 3.6 is defined as the ratio of the aborted existing sessions to the total accepted sessions, the new session drop rate is defined as the ratio of dropped new sessions over the generated sessions, and the completed session ratio is the completed number of sessions

³Experiments with TPC-W's shopping and browsing mixes were also conducted. Results are qualitatively the same as with the ordering mix and are not reported here due to lack of space.

divided by the generated sessions.

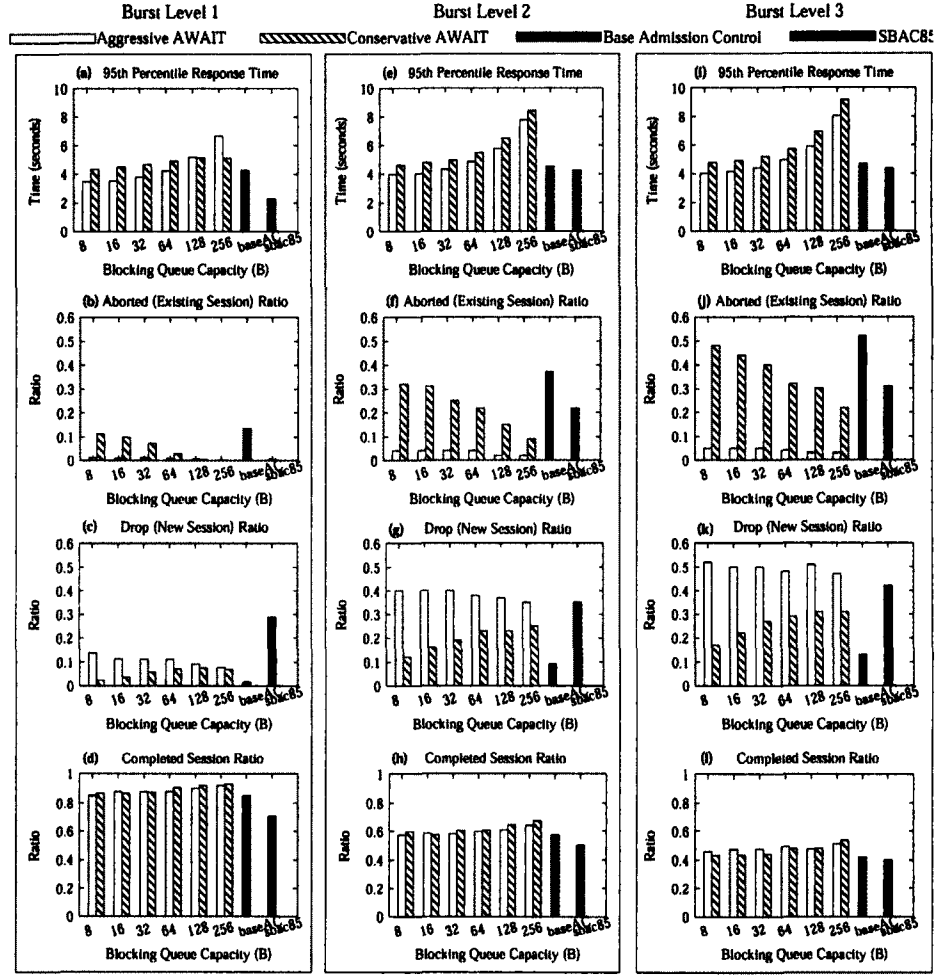


Figure 3.6: AWAIT with fixed size of the blocking queue. The graphs illustrate performance values for the aggressive and conservative versions (see white and shaded bars, respectively) for various fixed sizes of the blocking queue B . In all experiments, the limit of accepted requests A is set to 256, based on capacity planning.

The figure presents results for the three burstiness profiles in the arrivals of new sessions. First, one can easily see that the degree of burstiness in the arrivals dramatically impacts the user perceived performance, see the 95th percentiles of user response times for the various policies, see first row of graphs in Figure 3.6. Looking just at the percentiles, it is clear that the addition of the blocking queue deteriorates the user end-to-end times but the real benefit of blocking can be seen in the decrease of the aborted session ratio, see the second row of graphs, as well as in the decrease of new session drop ratio, see the third row of graphs. The useful

throughput of the system (measured in successfully completed sessions) is shown in the last row of graph that demonstrate the improved metric for both versions of *AWAIT* strategy compared to SBAC and *baseAC*.

Under low burstiness conditions, see first column of graphs, it is apparent that SBAC remains a good choice, at the expense of a very high percentage (nearly as high as 30%) of new session rejections. The aggressive and conservative versions of *AWAIT* result in longer response times but in significantly lower drop ratios, see Figure 3.6(b).

The effectiveness of the aggressive version to keep the aborted session ratio low is apparent across all burstiness levels, see Figures 3.6(b), 3.6(f), and 3.6(j) (second row of graphs). These figures show that the aggressive version very effectively differentiates between existing and new sessions, and treats existing sessions preferentially.

Naturally, because of the limited system capacity, if the number of accepted sessions that are aborted is low, then the ratio of rejected new sessions is bound to increase. This effect is shown for the aggressive policy in the third row of graphs in Figure 3.6, but this is unavoidable since our purpose is to bias the system for processing the requests of already accepted sessions against admitting new sessions, especially under periods of bursty traffic.

There is an additional question on the effectiveness of the aggressive *AWAIT* strategy compared to its conservative version: whether "flushing" the system queues might result in a less efficient resource usage and potentially may lead to a lower useful throughput. The last row of graphs in Figure 3.6 answers this question. It shows that the useful throughput of the system measured in successfully completed sessions is very similar for both conservative and aggressive versions of *AWAIT* and also significantly higher than SBAC or the simple *baseAC* policy. Overall, Figure 3.6 argues for the effectiveness of the aggressive version of *AWAIT*. In

the remaining of this chapter we focus on how to provide an autonomic version of *AWAIT* that adapts its configuration parameters to the workload. Before we move into the adaptive version, we examine the existing results more closely.

3.2.3 Performance Effect of the Blocking Queue Size

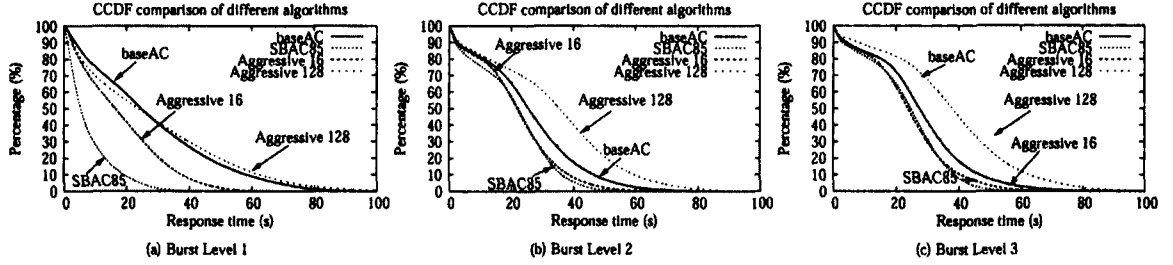


Figure 3.7: CCDF of system response time for different strategies for aggressive *AWAIT* that operates with a blocking queue size of 16 slots and a blocking queue size has 128 slots.

One of the special goals we pursue in this chapter is the overload management design that can support the application SLO requirements. Figure 3.7 shows the detailed latency profiling for all four strategies under study and different traffic burstiness. The graphs in Figure 3.7 present the complementary cumulative distribution function (CCDF) that helps in understanding how often the random variable (response time in our study) is above a particular value. We focus on *AWAIT* with two different blocking queue sizes equal to 16 and 128.

The figure confirms that increasing the blocking queue capacity leads to a significant increase in the latency of completed sessions, especially for arrivals with higher burstiness level. For smaller size of the blocking queue and higher levels of burstiness there is less difference between all the four strategies, and the latencies of completed sessions are closer in their profiles to the *baseAC* strategy. However, the larger blocking queue capacity changes the behavior of the underlying system in a significant way leading to the higher throughput of completed sessions but at a price of their higher latency. These results do stress the importance of correct

sizing of the blocking queue capacity. The workload burstiness combined with the size of the blocking queue have a critical impact on the request latency. If we aim to build an efficient overload management mechanism, then it should adapt its behavior to take into account traffic burstiness and to tune appropriately the blocking queue size.

3.2.4 Handle the Effects of Bottleneck Switch

Typically, the resources of the front tier present a bottleneck in the multi-tier system. In these cases, the usage-based admission control, applied to the front tier, provides a reasonable protection against overload. For example, the original SBAC that was proposed for a single-tier web server can be adopted for the multi-tier system in such a way that it allows accepting a new session only when the front server CPU utilization is below a certain threshold. In Section 3.1, we analyzed the SBAC performance for the multi-tier system that is processing the ordering transaction mix of the TPC-W benchmark. Under this workload, the front server is the system bottleneck. With the utilization threshold set to 85% and 95% respectively, SBAC provides a good overload protection for workloads without burstiness or low level burstiness. However, for higher levels of burstiness, the simulation results show that SBAC becomes quite inefficient as a protection mechanism and leads to a significant ratio of aborted sessions. This was the motivation to search for the alternative admission control mechanisms and introduce *AWAIT*.

The question is how sensitive the *AWAIT* strategy becomes to a change of the bottleneck in the system? How robust is *AWAIT*'s performance in case that the back-end server (and not the front server or front servers) is the primary system bottleneck?

Figure 3.8 shows the SBAC strategy performance (with the CPU utilization threshold of the front server set to 85% and 95% respectively) under a scenario when

the back-end server is being a primary system bottleneck, i.e., when the back-end server becomes highly overloaded with the increased load. The performance of SBAC is compared to the simple admission control strategy, called *baseAC* (see Section 3.1 for more details). The *baseAC* strategy employs a traditional overload control based on admitting a fixed, predefined number of active requests for processing (the number of active requests is set to 256 as suggested by capacity planning described in Section 3.1).

As we can see from Figure 3.8 that the SBAC strategy loses its performance advantages which we observed earlier in Figure 3.4. Now its performance is practically the same as the *baseAC* strategy which is regulated by a fixed number of active requests in the system. As Figure 3.8 shows that all metrics for *baseAC*, SBAC85% and SBAC95% are similar: the 95th percentiles of the user response time shown in Figure 3.8 (a) are identical for all three strategies, the ratios of aborted sessions coincide for the considered three strategies (see Figure 3.8 (b)), and the ratios of rejected new sessions (Figure 3.8 (c)) are the same.

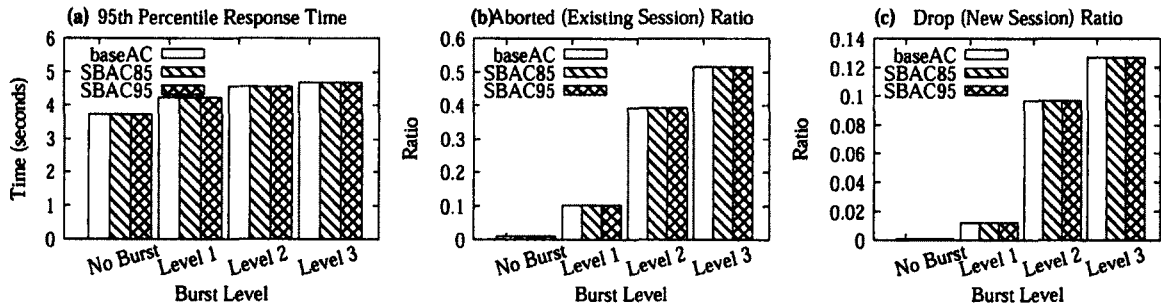


Figure 3.8: SBAC performance under the three different burstiness profiles with highly overloaded back-end server.

The explanation of this changed behavior is straightforward: it is due to the bottleneck switch from the front server to the back-end tier. A single-tier admission control like SBAC is usage-based: it is checking the front server utilization level to determine whether to accept a new session request or not. When the front server is not a bottleneck resource it always has a CPU utilization below 85%, and the

back-end server gets highly overloaded. In this case, the usefulness of a single-tier admission control policy like SBAC is severely diminished.

Now, our question is whether the proposed *AWAIT* strategy is robust to the bottleneck switch and whether it performs equally well when different tiers in the system represent the resource bottleneck. Figure 3.9 shows the performance of the aggressive of *AWAIT* as a function of the capacity of the blocking queue B with the back-end server being highly overloaded.

It is apparent that the proposed *AWAIT* strategy is robust to the bottleneck switch phenomenon, as it bases its decisions on the number of *ActiveRequests*, which is defined at the whole system level. Naturally, a larger capacity of the blocking queue leads to a smaller fraction of aborted sessions as well as to a smaller ratio of rejected new sessions. Figure 3.6 shows that *completed session ratio* (4th row) is related to both *aborted ratio* (2nd row) and *new session drop ratio* (3rd row). *AWAIT* significantly reduces the aborted ratio at the cost of an increased new session drop ratio, as desired. In general, the larger capacity of the blocking queue improves the ratio of completed sessions, but at a price of the higher values of 95th percentile of the response time and the increased average response time that is shown with the horizontal line inside the bars (see the first row of graphs). Overall, the results highlight the fact that the effectiveness of *AWAIT* is related to the blocking queue size.

3.3 Autonomic *AWAIT*

Here, we show how we can adjust on-the-fly the size of the blocking queue B in order to achieve a certain predefined SLO. To dynamically adjust the blocking queue size, we use both historical information of the achieved 95th percentiles of all requests served by the system (irrespective of the blocking queue capacity used --

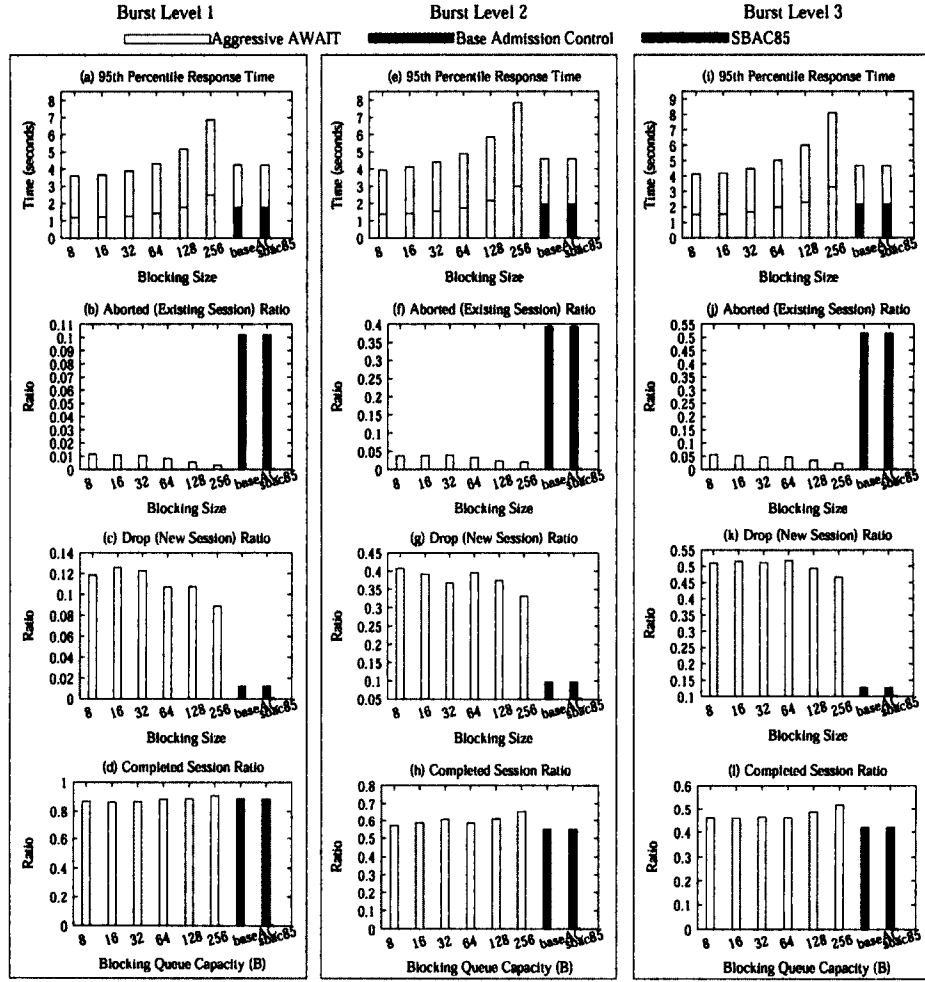


Figure 3.9: Aggressive AWAIT: strategy performance for various fixed sizes of the blocking queue B with highly overloaded back-end server. In the first row, the horizontal lines inside the bars reflect the average request processing time.

this value should reflect the target system SLO as the size of the blocking queue is transparent to the user) and response time percentiles that correspond to every other blocking queue capacity B used since the inception of the system. We use this information to decide whether the current blocking queue capacity is sufficient or not. Changing the blocking queue capacity B throughout the lifetime of the system is critical as during workload surges smaller B 's result in better performance rather than large B 's.⁴ To make the values of the 95th percentiles of the

⁴This may initially seem counter-intuitive as workload surges would result in large numbers of requests that exist simultaneously in the system. However, in order to maintain the target SLOs during a surge it is necessary to limit the blocking queue capacity, otherwise the time spent there dominates user response times and SLOs are violated.

user response times readily available, we maintain for each blocking capacity B a corresponding histogram of the user response times for that B . Therefore, for each completed request, two response time histograms are updated: the histogram of all requests in the system (irrespective of the blocking capacity B) and the histogram that corresponds to the current block capacity B used.

We change the capacity of the blocking queue for every group of $K = 10,000$ requests served.⁵ The autonomic algorithm then compares the achieved response time percentiles of *all* jobs in the system and the response times percentiles of the current configuration B with the target SLOs. If both percentiles are less than the SLO and there are aborted sessions, then it is clear that we can reduce the aborted ratio because there is room to increase B (since response times percentiles do not violate the SLO). If both percentiles are greater than the SLO, then the blocking queue should be reduced in an effort to meet the SLO target. If none of the above two conditions are met, we opt to leave the blocking queue capacity in its current level, otherwise the system may suffer from thrashing. For example, if the response time percentile of all requests is violated, but the percentile of the current B is not, the algorithm still stays with the current blocking queue size B , since the system is on a positive state and its accumulated statistics eventually correct the percentile of all requests.

The steps of increase/decrease of the blocking queue capacity can be arbitrary. In the experiments presented in this section, the capacity of the blocking queue B can have sizes as small as 1 and as large as 120. The increase/decrease step is equal to 5 for values of B less than 10 and equal to 20 for values of B greater than 20. We stress that other step values could also work, their selection affects how quickly the algorithm converges to a desirable B range. Algorithm 3.2 summarizes

⁵We selected $K = 10,000$ to be able to collect meaningful statistics for a group of requests. We performed a sensitivity analysis with the K value varying from 5,000 to 15,000. The results shows that the performance is nearly insensitive to the setting of K .

the policy.

Algorithm 3.2: Autonomic AWAIT: algorithm for changing the blocking queue size B on-line.

```
for every aborted session do
  AbortedSessions++;

for every finished request do
  counter ++;
  update total_RT_histogram (all requests, irrespective of  $B$ );
  update current_ $B$ _RT_histogram (with current blocking queue  $B$ );
  if counter ==  $K$  then
    if total_RT_percentile < SLO and current_ $B$ _RT_percentile < SLO
      and AbortedSessions > 0 then
        // Reduce aborted ratio
        increase current blocking capacity  $B$ ;
    if total_RT_percentile > SLO and current_ $B$ _RT_percentile > SLO
      then
        // Meet SLO target
        reduce current blocking capacity  $B$ ;
    counter ← 0;
    AbortedSessions ← 0;
```

The effectiveness of the autonomic AWAIT strategy is illustrated in Figure 3.10. Here, we experimented with the three different burst levels but also using different target SLOs. The figure illustrates how the blocking queue size changes as a function of the number of requests that are processed by the system for the various experiments. In each graph we also report on the achieved 95th percentile of the response time, as well as on the aborted and new session drop ratios. The figure shows that the autonomic AWAIT is remarkably robust: it reaches the target SLOs exceptionally well for all cases, while maintaining very low aborted rates. For each burst level, as the target SLO increases, the algorithm effectively increases the blocking queue capacity while reducing the aborted ratio. If we maintain the same SLO but change the burstiness of arrivals, the algorithm decreases the capacity of the blocking queue B . In all experiments, requests from existing sessions are pref-

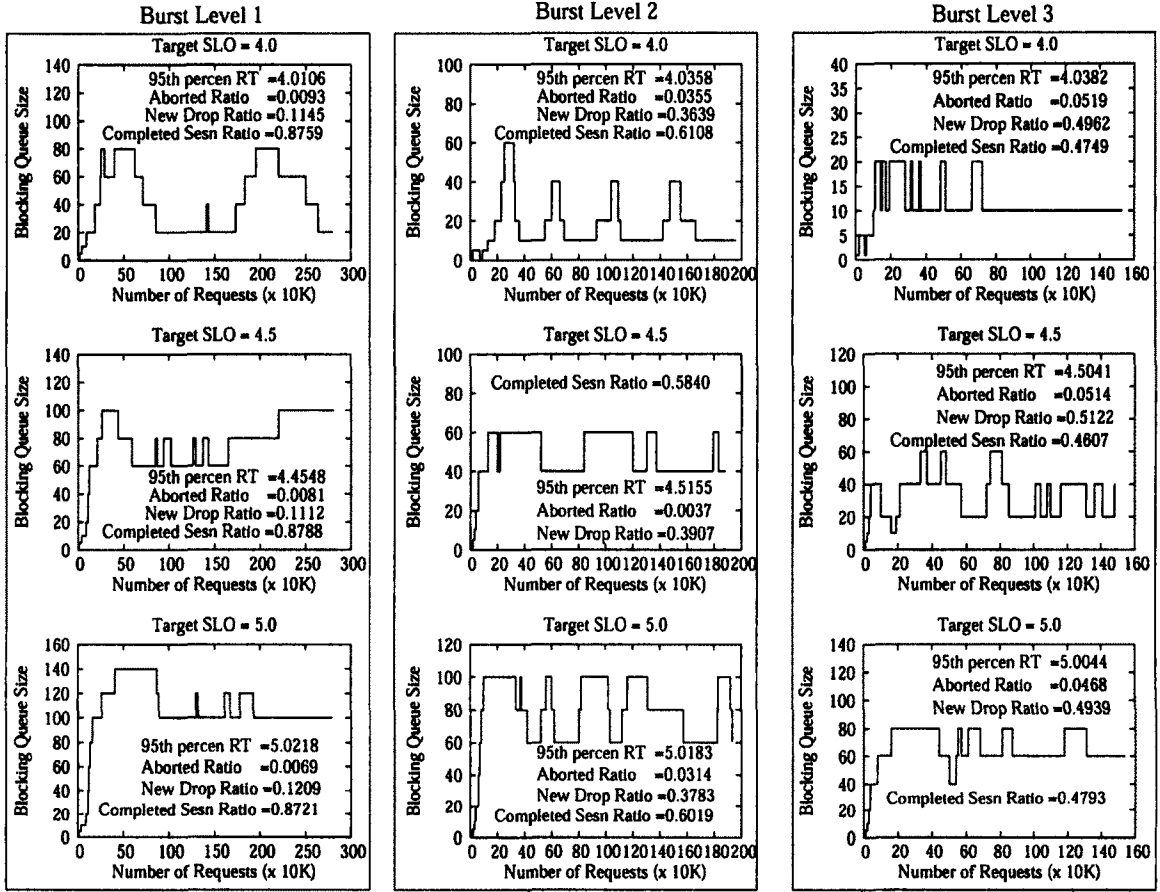


Figure 3.10: Autonomic AWAIT: illustration of how the capacity of the blocking queue B changes as a function of the workload.

erentially treated as low aborted ratios across all experiments are reported, and the ratio of successfully completed sessions is higher under the autonomic AWAIT policy compared to the aggressive static AWAIT strategy introduced in Section 3.2.1. These results demonstrate the effectiveness and robustness of the proposed autonomic mechanism of the aggressive AWAIT policy.

Note that the target SLO can be achieved with a fixed blocking size queue, but the size of the blocking queue needs to be different depending on the degree of burstiness (e.g., $SLO = 4$ seconds can be achieved with a blocking queue size set to 8 for the burst levels 2 and 3, but if the system operates under burst level 1, then the blocking queue size could be set to 32, see Figure 3.6). Any fixed configuration does not adapt to a changing traffic pattern. The proposed autonomic

strategy is specially designed to ``auto-tune" the blocking queue size for achieving and supporting a given SLO in the most optimal way.

In the following experiment, we designed a special workload that goes through different request arrival patterns. Initially, the arrival process starts with burst level 3, after that it is followed by burst level 1, and finally it follows the pattern defined by burst level 2. The overall workload arrival pattern is shown in Figure 3.11 (a).

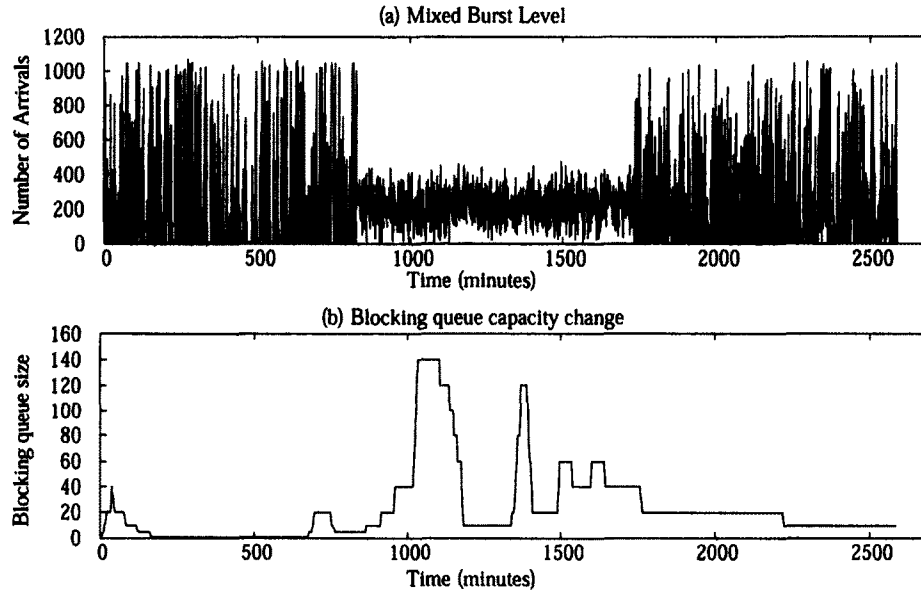


Figure 3.11: (a) Arrival process with different burstiness levels; (b) Blocking queue capacity changes as a function of the workload.

For this experiment, we set the SLO target to be 4.0 seconds and examine how well the autonomic version of *AWAIT* respects this target but also how it adjusts the capacity of the blocking queue as a function of the observed workload pattern. Figure 3.11 (b) reports how autonomic *AWAIT* changes the size of the blocking queue as the burstiness in arrivals changes: the policy reduces the size of the blocking queue to small values (as low as 1) after entering the burst level 3 time period, dramatically increasing it during level 1, and reducing it back to 10 after burst level 2. The autonomic strategy reaches a 95th percentile of response time equal to 4.03 seconds (perfectly on target), aborted ratio equal to 0.0304, and new

session drop ratio equal to 0.336.

To compare the effectiveness of the autonomic version comparing to the aggressive *AWAIT* with fixed blocking queue capacity, we experiment with the arrival process varies as shown in Figure 3.11 (a) and the aggressive version with the blocking size fixed and set to 8 (i.e., the best configuration for burst level 3 to meet the SLO of 4.0 seconds, see Figure 3.6), and to 32 (i.e., the best configuration for burst level 1, see Figure 3.6). For these experiments, the aggressive *AWAIT* with queue capacity 8, the 95-th percentile of response time is 3.81 seconds, the aborted session ratio is 0.0319, and the new session drop ratio is 0.345. With queue capacity equal to 32, the 95-th percentile of response time is 4.18 seconds (which is above the desirable SLO target), the aborted session ratio is 0.0286, and the new session drop ratio is 0.347.

To closely examine the performance values of the three policies, we show in Figure 3.12 the accumulated moving averages of the three metrics of interest. Fixing the blocking queue to 32 results in clear and constant violations of the SLO, see last column of graphs. A blocking queue set to 8 is effective at the beginning of the time, but then clearly stays well below the SLO target with the expense of a higher aborted ratio. The flexibility of dynamically adjusting the queue capacity to match the incoming workload is clearly shown on the improved values of the autonomic version (left column of graphs in Figure 3.12), where all metrics of interest are clearly in favor of autonomic. These experiments further corroborate that the proposed autonomic *AWAIT* strategy indeed manages to automatically adjust the blocking queue capacity to meet the desirable SLO targets by taking into account the observed pattern of arrival process, the system behavior, and the target performance metrics.

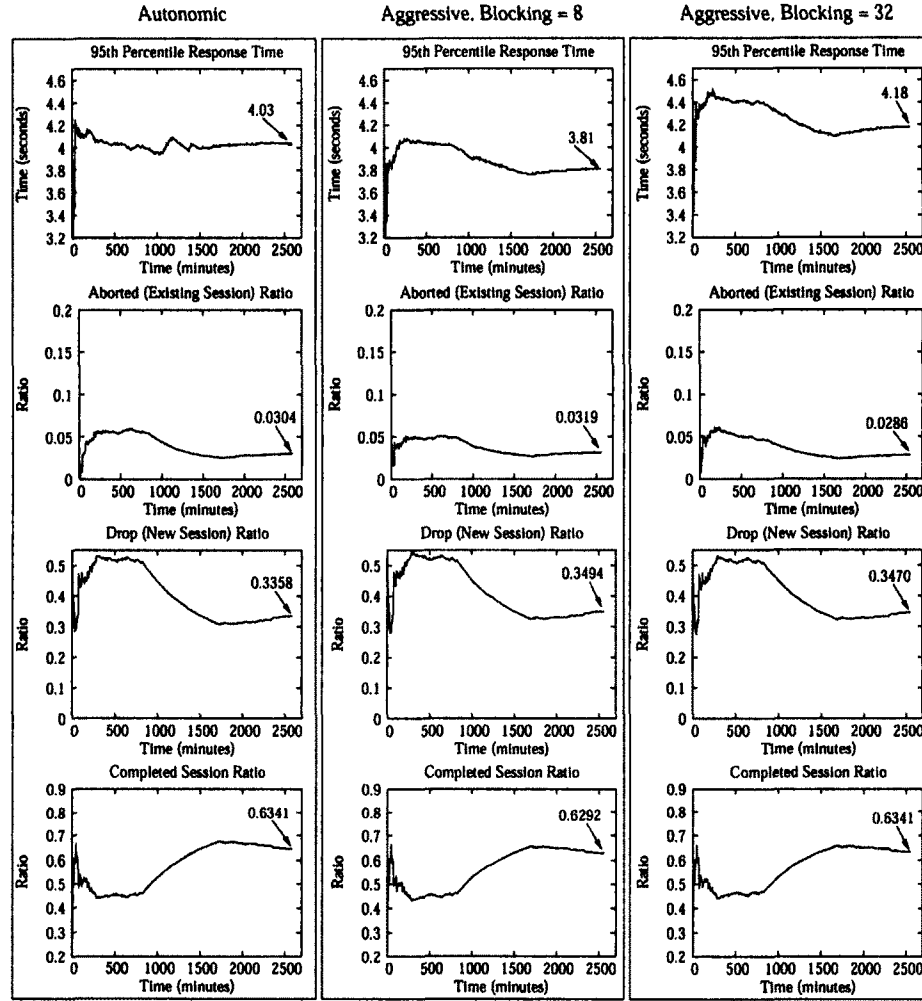


Figure 3.12: Moving 95th percentile of response time and moving average of aborted ratio, drop ratio and completed session ratio under the request arrival pattern shown in Figure 3.11 (a).

3.4 Comparisons with an Approach Based on Control Theory

Control-theory approaches have been proposed as an alternative way to maintain quality of service targets in web environments. In this section we evaluate *AWAIT* versus a classic, control-theoretic approach first proposed in [29]. Note that this approach focuses on a single tier as well. Our analysis focuses on its effectiveness but also highlights the fact that if the phenomenon of persistent bottleneck switch

is present due to burstiness, then single-tier techniques are clearly not effective.

The feedback-based approach given in [29] can be summarized as follows. Utilization control is central to the algorithm in [29]. The server's utilization is aimed to be kept at or below the target utilization value of $U^* = 0.58$, a "hard coded" value that has been proved to be effective in meeting the deadline constraints of real-time systems [29]. To maintain this target utilization, the server is able to offer "degraded" service levels in addition to the normal service level. Rejection can be considered as an extreme degradation point, at which the client receives no service. The service contents are pre-processed and stored in multiple copies that differ in quality and size. For example, a URL, such as, "my_picture.jpg" can be served from either "full_content/my_picture.jpg" or "degraded_content/my_picture.jpg" depending on the load conditions. In general, a server has M discrete service levels, numbered from 1 to M in increasing order of quality for the same content, while a service level of 0 means that the request is rejected.

The control variable m is an indicator of current service levels offered to clients and is in the range $[0, M]$. If m is an integer, all clients are served with level m . If m is a real number, then two levels of service are provided: $\lfloor m \rfloor$ and $\lfloor m + 1 \rfloor$. More specifically, a fraction F of the requests is served at level $\lfloor m + 1 \rfloor$, and $1 - F$ of the requests are served at level $\lfloor m \rfloor$. The algorithm periodically monitors the current utilization (U) and measures the "utilization error" $E = U^* - U$. It uses the well-known integral controller to produce the control output m . At each sampling time the controller performs the following computation:

$m = m + kE$; **If** ($m < 0$) **then** $m = 0$; **If** ($m > M$) **then** $m = M$; where k is a constant equal to 0.5

In the context of this work, M is set to 1 because only one service level is provided, i.e., m can have two possible values, 0 and 1. Here, we use the front server to measure its utilization and drive the algorithm. We experiment with two different

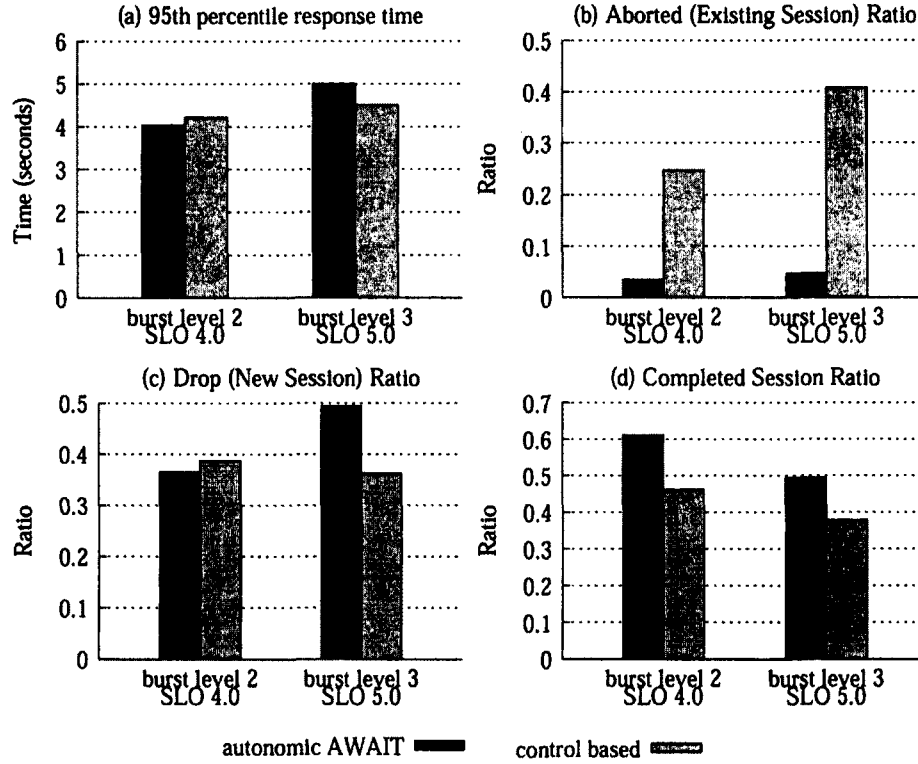


Figure 3.13: Comparison of performance of autonomic AWAIT and the control theory-based algorithm developed in [29].

burst levels with two different SLOs: burst level 2 with SLO equal to 4.0 and burst level 3 with SLO equal to 5.0. Figure 3.13(a) illustrates the comparative performance of AWAIT and the algorithm in [29]. In both experiments, AWAIT does not miss its SLO target. The control-based algorithm almost misses the SLO equal to 4.0 target in the first experiment but in both experiments it is not effective in differentiating the existing versus new sessions, see the high aborted ratio of existing sessions in Figure 3.13(b) and the relatively low ratio of new sessions in Figure 3.13(c). Meanwhile, the useful throughput of the system measured in completed session ratio of AWAIT is clearly significantly higher than that of the control-theory based algorithm, see Figure 3.13(d).

Overall, Figure 3.13 argues for the effectiveness of AWAIT versus a classic control theory approach and also highlights the fact that existing autonomic controllers that may work well in single-tier systems are ineffective in multi-tiered cases. Es-

pecially under bursty workloads, where the phenomenon of persistent bottleneck switch is present, there is a clear need to address the overload problem across all servers and not only within a single one. *AWAIT* clearly succeeds from this perspective.

3.5 Summary

This chapter presented an autonomic policy for service differentiation and admission control during overload for multi-tier system management. We focused on the pitfalls of existing policies under bursty conditions and remedy the problem by proposing the concept of a blocking queue where requests from already accepted sessions are stored if the system operates under overload. A novel autonomic algorithm, called *AWAIT* is proposed. *AWAIT* can limit the increase of the end-to-end response times within predefined SLO targets while dynamically adjusting the capacity of the blocking queue to the workload burstiness. Detailed simulations with the widely used TPC-W e-commerce benchmark under a variety of workload burstiness levels support the effectiveness and robustness of *AWAIT*.

The current algorithm adapts the blocking queue capacity to shield the offered web service from bursty arrivals, to provide service differentiation, and to prevent the system from overload. It complements the basic overload mechanism that sets a limit on the number of active client requests that are simultaneously processed by the system. Currently, this limit is defined by capacity planning.

4 Calibrating Resource Utilization in VMs

Autonomic resource management becomes increasingly required in large-scale computing systems for reducing administrator burdens. Autonomic resource management and system capacity planning often rely on analytic performance models [72, 76, 130]. Correct parameterization of such models is critical for their effectiveness [130]. Server virtualization clearly enhances flexibility in resource control, but introduces new challenges in parameterization of performance models. The relationship between application workload and physical resource utilization can be greatly obscured by the virtualization layer.

This chapter addresses a fundamental problem in virtual machine resource management: how to effectively profile physical resource utilization of individual VMs. Our focus is not just on collecting usage statistics but on extracting the utilization of physical resources by a VM across time, where the resources include CPU (utilization in CPU cycles), memory (utilization in memory size), network (utilization in traffic volume), and disks (utilization in disk I/Os). Correct VM resource utilization information is tremendously important in any autonomic resource management that is model based. For example, in dynamic provisioning, correct per-VM resource utilization information is the basis for the right VM sizing decision; in application management, performance modeling requires correct per-VM resource

utilization information to build the relationship between application performance and resource demands.

Profiling is a hard problem because mapping virtual-to-physical (V2P) resource activity mapping is not always one to one and may depend on application workload characteristics. The problem is further exacerbated by cross-resource utilization causality among different resources due to virtualization and multiplexing among VMs in a consolidated environment.

Here, we formulate the VM resource utilization profiling as a source separation problem, which is originally studied in digital signal processing. The aggregate utilization information of one physical resource is viewed as a mixed signal superimposed by the utilization signals of every individual VM. The objective is to figure out what are the original per-VM "signals". In this chapter we extend the factor graph model [83] with directionality and factoring generalization, and design a *directed factor graph* (DFG) that models the multivariate dependence relationships among different resources and across virtual and physical layers.

To build the base DFG model, we first focus on building separately DFG sub-graphs using micro-benchmarks and benchmark applications that are CPU-intensive (SPEC CPU 2006 [15]), memory-intensive (SPEC CPU2006), network-intensive (Netperf [12]) and disk I/O-intensive (IOzone [8], SysBench [16]). We also design a run-time calibration mechanism which outputs physical resource utilization estimation on individual VMs based on monitoring information and the DFG based model. The run-time calibration mechanism also includes a robust remodeling process that can make a new guided regression model to adapt to the temporal dynamics in the modeled resource relationships.

We use the Xen virtualization environment and apply the calibration mechanism on a set of consolidated VMs hosting diverse applications including RUBiS (a 3-tier app), Netperf, IOzone, and SysBench. The VM resource calibration output

is compared with the ``baseline" case defined as the physical resource utilization when that VM is hosted *alone* on the same server and with the same application workload. The results show that the calibration mechanism significantly improves the accuracy of the resource utilization information that are collected within guest VMs, reducing the relative error in CPU utilization from 44.8% down to 3.9% for CPU-intensive applications, and the relative error in disk write rate from 383% down to 16.7% for IO-intensive applications.

4.1 Problem Formulation

In this section, we first present background to Xen virtualization and then report a few cases of representative VM measurement information mismatching that motivate us for the development of a reliable information calibration approach. Then, we present the problem formulation.

4.1.1 Xen Virtualization

Xen [34] is an open source x86 virtual machine monitor which can create multiple virtual machines on a physical server. Each virtual machine runs an instance of an operating system. A scheduler is running on the Xen hypervisor to schedule virtual machines on the processors. Domain-0 in Xen is a privileged control domain used to manage other domains and resource allocation policies.

Xen does not account for resource consumption in the hypervisor on behalf of an individual VM, e.g., for I/O processing. On Xen's I/O model, a special privileged virtual machine called driver domain (by default ``Domain-0") hosts unmodified device drivers and directly controls physical devices. Other virtual machines, called guest domains in Xen, have to communicate through the driver domain to access the devices (e.g., network cards or disks). This I/O model results in a complex resource

utilization model. For example, an IO-intensive application has two components in its CPU utilization: CPU consumed by the guest domain where the application runs and CPU consumed by the driver domain which performs I/O processing on behalf of the guest domain. When multiple VMs are co-hosted on a single physical server, a problem posed by the Xen I/O model is to classify the driver domain's CPU consumption across the various guest domains. Similar problems arise for classifying the resource consumption of network and disk activities.

4.1.2 Information Mismatching Paradox

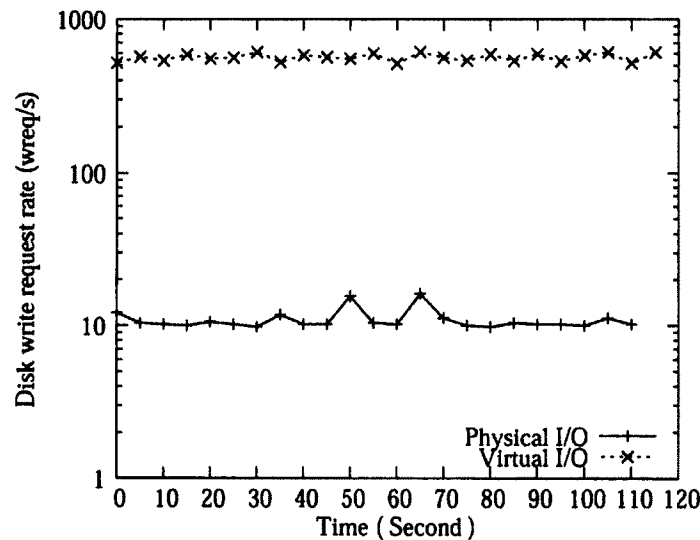


Figure 4.1: Measurement information mismatching: a disk I/O utilization example

Directly profiling VM's resource utilization inside the VM does not always give the correct information. For example, on a Xen-virtualized physical server hosting a single VM running IOzone [8] (a filesystem benchmark application), Figure 4.1 shows the disk I/O activities (write requests per second) measured inside the VM (called virtual I/Os) and the I/O activities measured on the physical disk (called physical I/Os). Both IO measurements are collected from the `/proc` file system in the guest domain and Domain-0 separately. There is more than one order of

magnitude difference between the two readings.¹

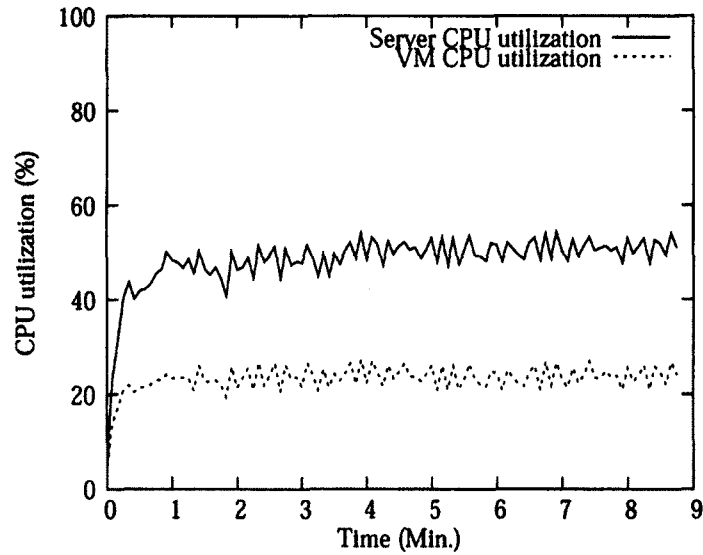


Figure 4.2: Measurement information mismatching: a CPU utilization example

Figure 4.2 shows another example on CPU utilization. On a Xen-virtualized physical server hosting a single VM running an Apache web server, we use the VM monitoring tool XenMon [73] to measure the VM's CPU utilization and the physical server's CPU utilization. While only a single VM is running, the server's CPU utilization is more than twice of the CPU utilization of the VM. This mismatching is mainly caused by the CPU overhead of Domain-0 in network and disk IO processing.

4.1.3 Problem Formulation

We define the problem as profiling physical resource utilization for an individual VM. That is, we want to profile how many physical resources have been utilized by each VM across time, where resources include CPU, memory, network, and disk.

¹Careful examination reveals that this is caused by write coalescence at the cache subsystem in the disk I/O layer.

4.1.3.1 Virtual Resource Monitored Information

Per-VM resource utilization information can be collected within the VM (e.g., via the `sar` utility tool) or from the VM Manager (e.g., Domain-0 of Xen). We implement the monitoring system to track various VM resource usage without modifying any virtual server:

- **CPU:** we monitor the consumed CPU by every individual guest VM. In Xen's Domain-0, CPU usage of guest VMs and Domain-0 itself are provided by the XenMon utility [73].
- **Memory:** we collect the memory usage as the ratio of used memory and the total memory allocated to the guest VM. While memory utilization is only known to the OS within each VM, tracking accesses to swap partitions from Domain-0 can infer such information [128].
- **Disk:** we collect the disk IOs issued from the guest VM to the privileged Domain-0 in four metrics - *wtps* (write requests per second), *bwrtn/s* (data written to vbd block device in blocks per second), *rtps* (read requests per second), *bread/s* (data read from vbd block device in blocks per second). In Domain-0 of Xen, such information for the guest VMs is available at `/sys/devices/xen-backend/vbd-<domid>-<devid>` for virtual block devices.
- **Network:** we collect the network traffic issued from guest VMs to Domain-0 in four metrics - *rxpck/s* (packets received per second), *txpck/s* (packets transmitted per second), *rxbyt/s* (bytes received per second), *txbyt/s* (bytes transmitted per second). In Domain-0 of Xen, such information on guest VMs is available in the proc filesystem at `/proc/net/dev` for virtual network devices.

4.1.3.2 Physical Resource Monitored Information

The following resource utilization information is collected at the privileged driver domain (e.g., Domain-0 of Xen):

- **CPU:** consumed CPU by the privileged domain.
- **Memory:** memory utilization as the ratio of used memory to total allocated memory of the privileged domain.
- **Disk:** four types of metrics are collected for aggregate physical IO that are the same as those for virtual disks.
- **Network:** four metrics are collected for aggregate traffic on physical network cards that are the same as those for virtual network devices.

4.2 Background Information

In this section, we describe the source separation problem defined in signal processing and a solution framework called factor graphs.

4.2.1 Source Separation

In digital signal processing, source separation problems [114] are those in which several signals have been mixed together and the objective is to find out what are the original signals. In particular, blind source separation is the source separation problem without any information about the source signals or the mixing process. Several approaches have been proposed for the solution of this problem type such as Singular Value Decomposition (SVD) and Principal Components Analysis (PCA). These approaches typically rely on the assumption that the source signals are mutually statistically independent. Unfortunately, such assumption does not

always hold in our application. For example, CPU overhead and network traffic originated from multiple VMs may have strong correlation when they belong to the same application services. VM disk write requests can be accumulated (delayed) and executed in batches on physical disks due to the page cache mechanism at the OS layer. Therefore, we focus on model based source separation approaches where domain knowledge on the mixing process can be encoded in the separation process.

4.2.2 Factor Graphs

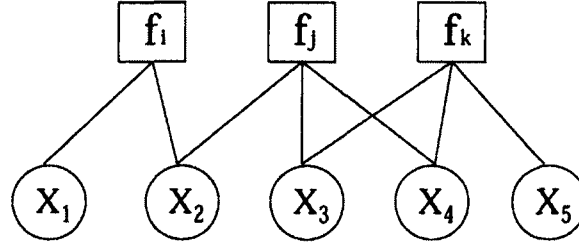


Figure 4.3: An example factor graph

Factor graphs [88] are graphical representations of complex mathematical models. They allow a unified approach to many source separation problems in signal processing and beyond. A factor graph is a bipartite graph representing the factorization of a global function of several variables. For example, assume that some global function, $f(x_1, x_2, x_3, x_4, x_5)$ can be factored as multiple local functions, e.g.,

$$f(x_1, x_2, x_3, x_4, x_5) = f_i(x_1, x_2) f_j(x_2, x_3, x_4) f_k(x_3, x_4, x_5)$$

This factorization is represented by the factor graph in Figure 4.3. In our application, we have to bring directionality into a factor graph so as to model a general decomposition of a global function into multiple local functions. We give details on this extension in the following section.

4.3 Directed Factor Graphs

In this section, we present the factor graph model that we use in the VM resource calibration problem.

4.3.1 Graph Model

Formally, a directed factor graph (DFG) is a bipartite digraph $G = (V, F, E)$. V and F are two disjoint node sets. V represent the set of variables, F represents the set of functions. One edge $x \rightarrow f$ in E connects a vertex x in V to one vertex f in F when x is an input parameter of the function represented by f . One edge $f \rightarrow y$ in E connects a vertex f in F to one y in V when y is an output parameter of the function represented by f .

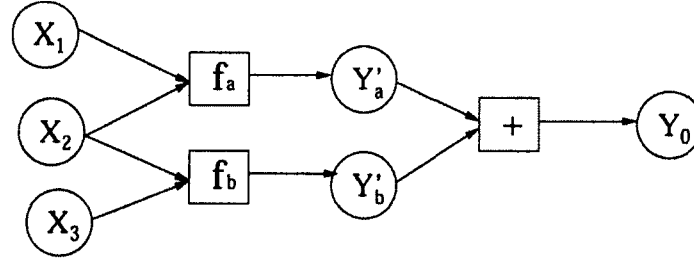


Figure 4.4: A directed factor graph example

Figure 4.4 shows the directed factor graph for a global function $Y_0 = g(x_1, x_2, x_3)$ with decomposition given as

$$g(x_1, x_2, x_3) = f_a(x_1, x_2) + f_b(x_2, x_3)$$

in Figure 4.4. The new variable nodes Y'_a, Y'_b are two temporary variables recording the output of the functions f_a and f_b .

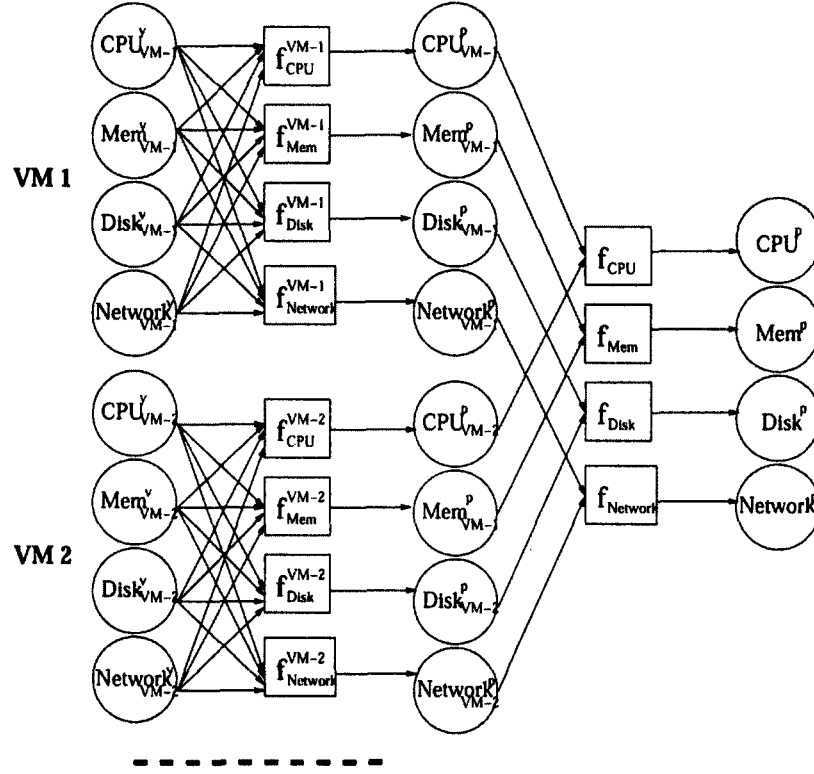


Figure 4.5: The directed factor graph in VM monitoring information calibration.

4.3.2 DFG in VM Information Calibration

We use the directed factor graph in Figure 4.5 as the base for VM information calibration. From left to right, the virtual resource activities are first transformed into the physical resource activities generated by each VM, and then are aggregated to render the physical resource activities of the hosting server. The left-most variable nodes represent observable virtual resource activities, the right-most variable nodes represent observable physical resource activities, see Section 4.1.3. The intermediate variable nodes, such as CPU^p_{VM-1} representing the physical CPU consumption by VM-1, are the data we want to infer and we derive them through statistical inference techniques on the function nodes such as f^{VM-1}_{CPU} .

We choose the DFG model in Figure 4.5 as it naturally describes the resource demand transformation and aggregation processes in a virtualization environment. The edges in the graph depict statistical causality relationships between resource

utilization at different components/layers. The generalization of this graph model is possible thanks to the flexibility in the identification of the function nodes, which may be different for different hypervisor architectures. In the following section, we show how to build a base model for Xen by identifying the function nodes through benchmark profiling and regression analysis.

4.4 DFG Based Model

In this section, we run different benchmark applications in one guest domain to generate the workload on each virtualized resource separately and to build the DFG based model for our calibration mechanism.

4.4.1 Methodology

The modeling process consists of the following steps:

1. Host a single VM in a server.
2. Run a benchmark for a specific virtual resource (e.g., a CPU-intensive benchmark).
3. Apply statistics analysis to find out the set of physical resources on which the benchmark incurs non-negligible utilization and learn the models for the function nodes such as f_{CPU}^{VM-1} .

The benchmark based modeling process aims at capturing the stable causality relationships between virtual and physical resource demands. We carefully select a fixed set of benchmark applications to cover all the four resources (CPU, memory, disk, and network) at the virtual layer. If the causality relationship changes across time, as for example in the disk IO access patterns of an application, then we resort to the guided regression method that is described in Section 6.2.

4.4.2 Regression Analysis

In Step 3, stepwise regression [58] is applied to the collected data to find out correlated measurement variables and to remove co-linearity that may exist between variables. Stepwise regression uses the same analytical optimization procedure as multiple regression but differs in that only a subset of predictor variables is selected sequentially from a group of predictors by means of statistical testing of hypotheses.

4.4.2.1 Source Node: Virtual CPU Load

We first run a micro-benchmark in the guest VM that alternates between sleeping and calculating Fibonacci numbers, the ratio of which determines the VM CPU utilization. Figure 4.6 shows the CPU overhead in the privileged domain while the micro-benchmark VM's CPU utilization changes from 5% to 50%. The Domain-0 CPU overhead is stable and remains close to 0. Figure 4.6 also shows the Domain-0 overhead when the guest VM runs *gromacs*, a CPU-bound SPEC CPU2006 benchmark. The Domain-0 overhead is close to 0 while the guest VM uses up its allocated CPU capacity (one core of the dual-core processor in the server). We also observe (not shown on this graph) that the CPU-intensive guest VM did not incur overhead on other server resources (e.g., network or disk).

4.4.2.2 Source Node: Virtual Memory Load

We examine how memory-intensive applications on guest domains impact the resource utilization in the privileged domain. Figure 4.6 shows the Domain-0 CPU overhead when running *libquantum*, a memory-bound SPEC CPU2006 benchmark, and the overhead is close to 0. We also observe (but not include these results here) that the memory-intensive guest VM did not impose overhead on other server resources (e.g. network or disk).

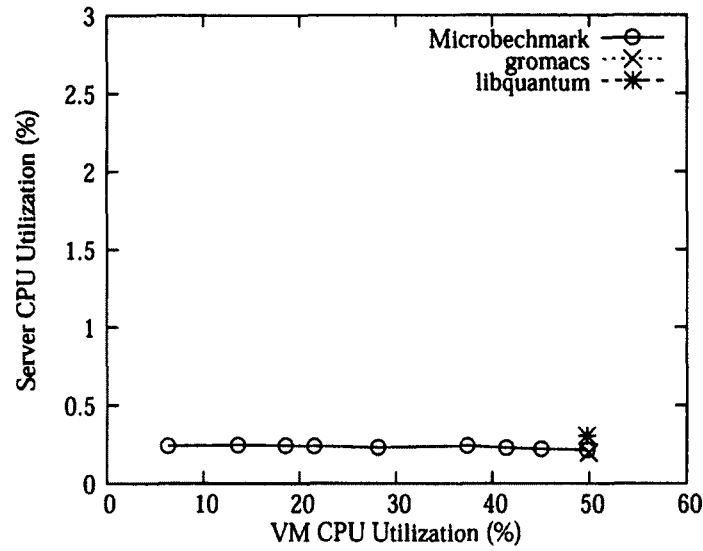


Figure 4.6: Compute intensive workload has no impact on privileged domain performance

4.4.2.3 Source Node: Virtual Network Load

We use Netperf [12] to perform controlled network load generation. Netperf allows both UDP and TCP stream tests. Here, we measure the number of packets (bytes) sent/received per second for both guest and privileged domains.

For the UDP case, we change the packet sending rate from 1,000 to 36,000 pkts/sec, the packet payload size from 100 bytes to 1,400 bytes, and measure server resource metrics on both sender and receiver sides. Since the TCP protocol is not allowed to specify the transfer speed, we can not change the sending rate settings as in the UDP protocol.

The regression coefficients of different metrics from stepwise regression algorithm is shown in Table 4.1, and only the correlated variables are presented. Each row corresponds to a Domain-0 resource metric and each column corresponds to a guest domain variable. For example, the first row in the figure means the privileged domain CPU overhead(%) = $6.64 \times 10^{-4} \times \text{recv_pkt_rate} + 5.02 \times 10^{-4} \times \text{send_pkt_rate} - 0.06$. It shows that the Domain-0 CPU overhead has a clear linear relationship with the packet sending and receiving rates, but not with the network

throughput in bytes. The same observation is also reported by Wood et al. [127].

Table 4.1: Network regression model.

| | | Domain-U | | | | |
|--------|---------|----------|----------|-----------|----------|-----------|
| | | rxbyt/s | rxpkt/s | txbyt/s | txpkt/s | intercept |
| Domain | CPU(%) | 0 | 6.64e-04 | 0 | 5.02e-04 | -0.06 |
| | rxbyt/s | 1.01 | 0 | 0 | 0 | -49.51 |
| | rxpkt/s | 2.24e-06 | 1.00 | 0 | 0 | 2.56 |
| | txbyt/s | 0 | 0 | 1.00 | 20.38 | 1094.2 |
| | txpkt/s | 0 | 0 | -4.14e-06 | 1.01 | 4.26 |

4.4.2.4 Source Node: Virtual Disk IO Load

Xen supports many different storage options for the guest domain. These options can be divided into three categories: *file based*, *device based*, and *LVM-based*. The file-based block devices can be differentiated by how Xen accesses them: *blktap* and *loopback*. Blktap replaces the common loopback driver for file-based images because it allows for improved performance and more versatile filesystem formats, such as QCOW [14]. It also avoids problems related to flushing dirty pages which are present in the Linux loopback driver.

We build our model for guest domains with file based disk storage. For the other two storage options, the same methodology can be also applied. Here, we perform controlled IO-intensive experiments with SysBench [16]. SysBench is a multi-threaded benchmark tool for evaluating database (e.g., MySQL) server performance under intensive load. We exploit its file IO performance testing functionality to generate different IO activities. To control the write/read operation rate, we add different sleeping time between each write/read operation in the source code. The SysBench IO testing module supports six IO operations: sequential write, sequential rewrite, sequential read, random read, random write, and combined random read/write. We take samples of each IO operation and create a data set based on all the samples. The block size is set to 16K bytes. The total size of testing files

is set to 4G bytes. We choose ``default'' for other option settings.

Tables 4.2 and 4.3 show the regression models extracted by blkmap based and loopback based devices. We note that besides the difference on CPU overhead, the relationship functions from virtual disk IOs to physical IOs are also different for blkmap based and loopback based devices. For example, the coefficient of virtual *rtps* to physical *rtps* is 1.29 for blkmap based devices, while the coefficient is 0.34 for loopback based devices. We observe (but not include the results here) that the regression models for disk IOs are dynamic and dependent on workload patterns (e.g., sequential vs random access, high vs low access locality). The coefficients in the regression models of Tables 4.2 and 4.3 represent the resource relationships under the standard SysBench workload. Later, we present a model relearning scheme for online information calibration that adapts to inevitable workload dynamics, see Section 4.5.

Table 4.2: Blkmap based device regression model

| | | Domain-U | | | | |
|--------|---------|----------|----------|------|----------|-----------|
| | | rtps | bread/s | wtps | bwrtn/s | intercept |
| Domain | CPU(%) | 0 | 2.02e-04 | 0 | 2.21e-04 | 0.47 |
| | rtps | 1.29 | 0 | 0 | 0 | 10.57 |
| | bread/s | 0 | 1.00 | 0 | 0 | 1.59 |
| | wtps | 0 | 0 | 0.11 | 0.0018 | 1.6 |
| | bwrtn/s | 0 | 0 | 0.07 | 0.998 | 15.3 |

Table 4.3: Loopback based device regression model

| | | Domain-U | | | | |
|--------|---------|----------|----------|------|----------|-----------|
| | | rtps | bread/s | wtps | bwrtn/s | intercept |
| Domain | CPU (%) | 0 | 8.27e-05 | 0 | 1.11e-04 | 0.23 |
| | rtps | 0.34 | 0 | 0 | 0 | 0.14 |
| | bread/s | 0 | 1.0 | 0 | 0 | -0.53 |
| | wtps | 0 | 0 | 0.10 | 0 | 1.52 |
| | bwrtn/s | 0 | 0 | 0 | 0.997 | 22.1 |

4.4.2.5 System Overhead

When multiple VMs are consolidated in one physical machine and lead to heavy IO utilization, possible system overhead needs to be considered . One example of the source of such overhead is the *ksoftirqd* daemon process. *ksoftirqd* is a per-CPU kernel thread that runs when the machine is under heavy soft-interrupt load. If a soft interrupt is triggered for a second time while soft interrupts are being handled, the *ksoftirqd* daemon is triggered to handle the soft interrupts in process context. The sudden run of *ksoftirqd* daemon process under heavy network workload could lead to unexpected CPU utilization bursts. In our solution, this type of overhead is taken as system noise and is excluded from the aggregate resource utilization contributed to guest VMs.

4.5 Information Calibration

In this section, we present the run-time calibration mechanism. The mechanism takes as input the VM monitoring information as described in Section 4.1.3 and outputs per VM physical resource utilization information based on the DFG model in Section 4.3.

4.5.1 Run-time Calibration Mechanism

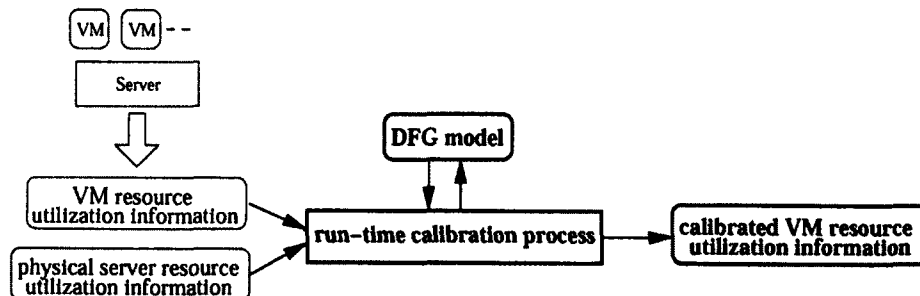


Figure 4.7: The run-time calibration mechanism

Figure 4.7 shows the overview of our run-time calibration mechanism. The inputs to the mechanism are the resource utilization monitoring information within the guest VMs and the privileged domain (for aggregate physical server load information). The mechanism uses the DFG model to decode the input information and outputs per-VM physical resource utilization information. In the run time, the calibration process may also update the DFG model if the existing one incurs non-trivial estimation errors.

We exemplify the algorithmic steps in the context of CPU utilization, but the same steps apply for all four resources. The run-time process on per-VM CPU utilization information calibration is given in Algorithm 4.1.

Algorithm 4.1: per-VM CPU utilization information calibration algorithm

- 0 Initialize the model parameters of DFG function nodes;
 - 1 Feed the DFG model with per-VM virtual resource utilization information;
 - 2 Calculate the value of DFG latent variables on per-VM CPU utilization information;
 - 3 Calculate the value of server CPU utilization variable;
 - 4 **while** $CPU_err > thresh$ **do**
 - 5 Re-learn the DFG function models;
 - 6 Re-calculate the value of the DFG latent variables for per-VM CPU utilization information;
 - 7 Output per-VM CPU utilization information;
-

In Step 0, the initial model parameters are obtained from benchmark based profiling, see Section 4.4, or from offline application-specific profiling for calibrated VMs. While the latter method is expected to give a more accurate model than the former one, it comes with an extra profiling overhead. In the evaluation, we use the benchmark based profiling results for this step.

Step 4 brings a feedback loop to make our calibration process adaptive to inevitable model dynamics, typically caused by the change of workload patterns. As shown in Section 4.4, the relationship between a virtual resource activity and its overhead on physical resources can vary and depend on the workload contents.

The mapping of virtual I/O to physical I/O activities is one such example. To be robust to transient workload changes or monitoring noise, the discrepancy is calculated on the average of the estimation errors during a sliding window including the past K time points. The threshold is chosen as $(\epsilon + Z_\alpha * \sigma)$, where (ϵ, σ^2) is the mean and variance of the regression model estimation error from the last remodeling process (or those learned from the benchmark based profiling at the beginning of the process). Z_α is the standard score in statistics [95], and here measures how unlikely an estimation error is if the current model is correct. If $Z_\alpha = 3$, $\alpha = 99.75\%$, then an estimation error larger than $(\epsilon + 3 * \sigma)$ is unlikely (with probability $< 0.25\%$) to appear if the virtualization environment were the same as during the last remodeling process. Therefore, if several large estimation errors in a row indicate the change of some factors in the virtualization environment, then a remodeling process is triggered. This process is presented in the following subsection.

4.5.2 Robust Remodeling: Guided Regression

While we use linear regression models in Section 4.4 for single VM based benchmark profiling, a new problem arises in the calibration process when multiple VMs are co-hosted in a single server: now \mathbf{y} represents a physical resource utilization which is the summation of physical resource utilization of multiple VMs. Since the physical resource utilization of each individual VM is a latent variable, a straightforward regression model is as follows, assuming m co-hosted VMs:

$$\begin{aligned} y^{(i)} = & (\beta_1^{VM-1} x_1^{VM-1(i)} + \dots + \beta_p^{VM-1} x_p^{VM-1(i)}) + \\ & \dots + (\beta_1^{VM-m} x_1^{VM-m(i)} + \dots + \beta_p^{VM-m} x_p^{VM-m(i)}) \end{aligned}$$

where $y^{VM-j(i)} = \beta_1^{VM-j} x_1^{VM-j(i)} + \dots + \beta_p^{VM-j} x_p^{VM-j(i)}$ is the latent variable for the j^{th} VM.

If we directly solve the above problem with the least square solution

$$\hat{\beta} = (\mathbf{x}^T \mathbf{x})^{-1} \mathbf{x}^T \mathbf{y}, \quad (4.1)$$

it could lead to re-learning the models of all the VMs in the server. We seek to enhance the original regression modeling method for remodeling robustness due to three reasons. The first comes from common run-time monitoring data error and noise (e.g., system noise, transient VM migration overhead) that might add transient perturbation onto otherwise stable resource relationships. The second is due to the factor that some relationships (such as virtual disk I/Os and its resource overhead) are naturally dynamic due to their content dependence. Re-learning those models should not affect the model of other stable relationships. The third reason is due to the fact that since the co-located VMs are all involved in the regression model, the number of unknown parameters in β is large. In order to obtain accurate estimation of those parameters, a significant amount of measurements $[\mathbf{x}, \mathbf{y}]$ is usually required. However, in the model relearning process, sometimes we do not have so many observations due to the quick dynamics of the system. The lack of (enough) data may lead to large variances of the final solution β .

In order to enhance the robustness of model estimation, we propose a guided regression process to solve the model of Eq. (4.1). We add some constraints to describe the range of possible β values and embed those constraints into the estimation process. The constraints can come from various sources, such as the prior model knowledge based on the benchmark profiling or the model learned during the previous time period. By including such knowledge to guide the estimation, we can obtain a more reliable solution β for the regression model.

The prior constraints on β are represented by a Gaussian distribution with the

mean $\bar{\beta}$ and covariance Σ

$$P(\beta|\sigma^2) = (\sigma^2)^{-K} \exp \left\{ -\frac{1}{2\sigma^2} (\beta - \bar{\beta})^T \Sigma^{-1} (\beta - \bar{\beta}) \right\} \quad (4.2)$$

The mean $\bar{\beta}$ represents the prior expectation on the values of β , and is determined from the β values learned in Section 4.4. The covariance Σ represents the confidence of our prior "knowledge". We choose Σ as a diagonal matrix $\Sigma = \text{diag}(c_1, c_2, \dots, c_p)$, in which the element c_i determines the level of variances of β_i in the prior distribution. If we are confident that the value of β_i is located closely around $\bar{\beta}$, the corresponding c_i value is small. Otherwise we choose large c_i values to describe the uncertainty of β_i values. Note that the least squares method in Eq (4.1) solves the regression without any prior knowledge, i.e., the values of c_i 's are infinite, which may be inaccurate when the number of collected measurements is insufficient.

There is also an unknown parameter σ^2 in Eq (4.2), which represents the variance of the data distributions. Here, we use the inverse-gamma function [95] to represent the distribution of σ^2 :

$$P(\sigma^2) = \frac{b^a}{\Gamma(a)} (\sigma^2)^{-(a+1)} \exp \left\{ -\frac{b}{\sigma^2} \right\} \quad (4.3)$$

where a, b are two parameters to control the shape and scale of the distribution, $\Gamma(a)$ is the gamma function of a . We choose the inverse-gamma function because: 1) it is one of the common distributions for non-negative variables such as the variance studied here; 2) it is easy to tune its shape by setting (a, b) parameters. 3) By using the inverse-gamma function as the prior of Gaussian variance, we can obtain a closed-form solution for optimizing the posterior distribution estimation.

Given the prior distribution $P(\beta)$, the guided regression finds the solution by

maximizing the following posterior distribution

$$P(\beta|\mathbf{x}, \mathbf{y}, \sigma^2) \propto P(\mathbf{y}|\beta, \mathbf{x})P(\beta|\sigma^2)P(\sigma^2) \quad (4.4)$$

which leads to the following solution

$$\beta^* = (\mathbf{x}^T \mathbf{x} + \Sigma^{-1})^{-1}(\Sigma^{-1} \bar{\beta} + \mathbf{x}^T \mathbf{x} \hat{\beta}). \quad (4.5)$$

Due to the space limit, we do not present the detail of the above derivations.

To summarize, the robust remodeling process takes the following steps: 1) decides the prior coefficients $\bar{\beta}$ and their weight metric Σ (e.g., learned through the benchmark profiling in Section 4.4); 2) solves Eq.(4.1) based on the run-time monitoring data for the standard least square solution; 3) calculates the final solution β^* , which is a weighted average of the two components from 1) and 2).

4.6 Evaluation

As discussed in the previous sections, the proposed DFG based model and run-time calibration mechanism constitute the two building blocks for VM resource utilization information calibration process. These building blocks can be directly applied to existing applications. In this section, we demonstrate three case studies that clearly show the effectiveness of the calibration methodology.

4.6.1 Experimental Methodology

We evaluate the effectiveness and accuracy of our calibration technique with different applications. The test-bed runs the Fedora release 8 operating system with Linux kernel 2.6.18-8. The evaluation is based on the Xen virtualization platform version 3.3.1. Our test-bed platform uses Supermicro 1U Superservers with Intel

Core 2Duo E4300 1.86 GHz processors, 2MB L2 cache. All servers have a RAM of 2GB and 250GB 5400RPM disk. The servers are connected through D-LINK DES-3226L 10/100Mbps switches. The test-bed is managed by Usher [96], an open source VM management middleware with a centralized monitoring database. The run-time calibration process is co-located with the monitoring DB and it calibrates the raw monitoring data in batches with fixed time window size.

The following applications are used in our evaluation:

- RUBiS is an auction site prototype modeled after eBay. A client workload generator emulates the behavior of users browsing and bidding on items. We use the Apache/EJB implementation of RUBiS version 1.4.3 with a MySQL database server version 5.0.77.
- IOzone is used for filesystem benchmarking. It is used to generate and measure various disk I/O activities.
- SysBench is a multi-threaded benchmark tool for evaluating database (e.g., MySQL) server performance under intensive load. We use SysBench to generate MySQL workloads which lead to various I/O activities.
- Netperf is tool for network benchmarking (see Section 4.4.2.3). We use Netperf to generate different network traffic workloads.

4.6.2 Results

Here, we present three scenarios to test the accuracy of the DFG based model. In scenario 1, we consider the RUBiS application. We collect baseline information from every component server i.e., web server, application server, and database server. We demonstrate substantial improvements in resource information estimation with the DFG based model for two different server consolidation environments.

In scenario 2, we illustrate how guided regression can be used to improve on the DFG based model. The estimation after the remodeling process shows dramatic improvements. In scenario 3, we illustrate how the DFG model is used to separate a mixed disk IO stream into its component streams.

4.6.2.1 Scenario 1: RUBiS

RUBiS is a multi-tier web service application composed by open-source software, i.e., Apache Web Server, JBoss EJB Server, and MySQL relational database. For all servers, we use blktp based virtual block devices. The resource usage baseline of each server is collected when it is virtualized and run in the same physical machine alone. Here, RUBiS is initialized with 700 simultaneous clients with the browsing workload. For the baseline comparison, we run RUBiS for 30 minutes and collect usage statistics.

We setup the following server consolidation environment (two physical machines): on Machine #1, the virtualized App server and DB server are consolidated together; the web server VM is placed on Machine #2. We collect each virtual and physical machine run-time information using the techniques described in Section 4.1.3. The calibrated usage information is calculated with the DFG based model described in Section 4.4.

Figure 4.8 illustrates the relative errors¹ of both in-VM monitoring and DFG based model. The figure shows results for all three component servers. We observe that RUBiS is mainly a compute and network IO intensive application with very low disk activities. In the figure, we only show metrics with significant values and ignore those close to zero. One can easily see that the DFG based model significantly reduces the relative error, e.g., the error in web server CPU utilization drops from 44.8% down to 3.9%.

¹Relative error is defined as the ratio of absolute error to its corresponding baseline value.

The substantial improvements in CPU utilization estimation is due to the fact that DFG based model takes the intensive network activity overhead into consideration. In the web server example, on the average, the server transmits about 12,500 packets per second to the clients and application server. Meanwhile, it also receives about 13,400 packets per second. According to the DFG based model, this amount of network traffic leads to 13% privileged domain CPU overhead that is not reported by in-VM monitoring.

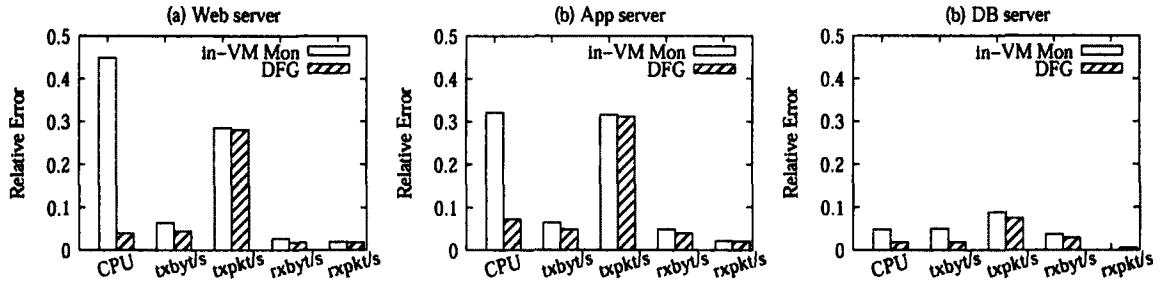


Figure 4.8: Relative error of in-VM monitoring method and DFG based model in RUBiS app

4.6.2.2 Scenario 2: Co-hosting Network- and IO-intensive Apps

For this experiment we consolidate two virtualized servers on one physical machine. Both VMs are configured with loopback based virtual block devices. The first VM runs Netperf that sends out UDP packets at the rate of 25Mbps and the second VM runs SysBench. SysBench is set to run in the ``oltp'' test mode, to emulate a real database. For our test-bed, we choose the MySQL implementation and set the number of rows in the testing table to 5,000,000. To make the testing more real, we select the execution mode to be ``advanced transactional'' in which each thread performs transactions.

In the experiment, we set a sliding window length to 15 minutes. At the beginning of the first window, the parameters of DFG are initialized according to Tables 4.1 and 4.3. The system is set to take monitoring samples every 10 seconds and report to the center database. According to the run time calibration mechanism in

section 4.5, at the end of each window, the program calculates the estimation error between the current DFG model and physical server measurement as shown in step 4 of Algorithm 4.1.

Table 4.4 shows the in-VM monitoring information, the DFG model result, and the resource usage of the physical server which is hosting the two VMs. We only report the metrics with significant values.

Table 4.4: Example of DFG error that triggers remodeling mechanism

| | CPU | txbyt/s | txpkt/s | wtps | bwrtn/s |
|-----------------|-------|----------|---------|---------|-----------------|
| in-VM1-mon | 27.45 | 0 | 0 | 2052.77 | 30859.36 |
| in-VM2-mon | 0.83 | 3.08e+06 | 3000 | 0 | 0 |
| DFG | 5.14 | 3.15e+06 | 3021 | 214.81 | 30773.46 |
| Server resource | 5.74 | 3.14e+06 | 3009 | 176.96 | 7927.70 |

The large estimation error highlighted in bold triggers the remodeling mechanism. Note that remodeling is only applied for the IO write metrics. The model of other resources is kept unchanged. The new model is based on the data points collected in the previous time window. The new disk IO write model replaces the original one. Close examination indicates that the large estimation error (see the bold value in the table) happens because that loopback based storage uses the Domain-0 kernel page cache. When a file write occurs, the page backing the particular block is looked up. If it is already found in cache, the write is done to that page in memory.

Figure 4.9 illustrates the relative error for in-VM monitoring method and run time calibration mechanism. Note the log-scale on the Y-axis. One can easily see the significant improvement in the estimation accuracy. The relative error of the ``bwrtn/s" is reduced from 276% down to 3.5%. The ``txpkt/s" metric in Figure 4.9(a) is still better than the calibration result after remodeling but the run time calibration has already given a good estimation which only has a relative error of 0.5% only.

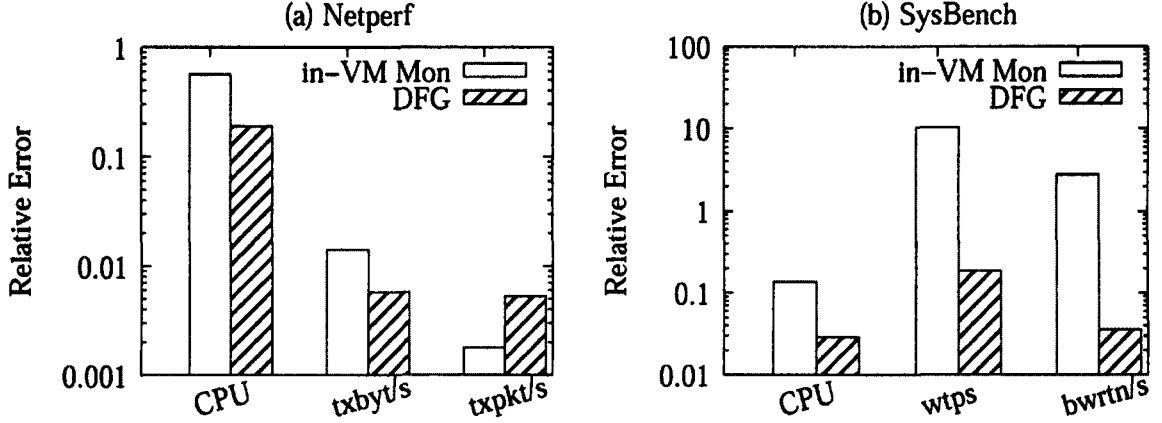


Figure 4.9: Relative error for in-VM monitoring method and run time calibration mechanism

4.6.2.3 Scenario 3: Co-hosting IO-intensive Apps

In scenario 3, we show the case study of how the DFG based model is useful in decomposing disk IO. We setup two virtualized servers on one physical machine. The first VM server executes the IOzone benchmark to perform only writes and re-writes. The second VM server runs the SysBench benchmark and writes to disk at the rate of 8MBps. The block size is set to 16K bytes and the total size of testing files is 8GB. We choose ``default'' for other options. In both cases, we use blktpas based virtual block devices.

The effectiveness of DFG based model in decomposing mixed IO and CPU is shown in Figure 4.10. The DFG based model successfully decomposes the mixed CPU utilization and reduces the relative error in write request rate from 391.5% down to 10.6% in Figure 4.10(a) and from 304.4% to 31% in Figure 4.10(b). Meanwhile, the figure shows that relative error in CPU is also reduced significantly.

4.7 Summary

In this chapter, we present the design and evaluation of a VM monitoring information calibration mechanism. We formulate the problem as a source separation

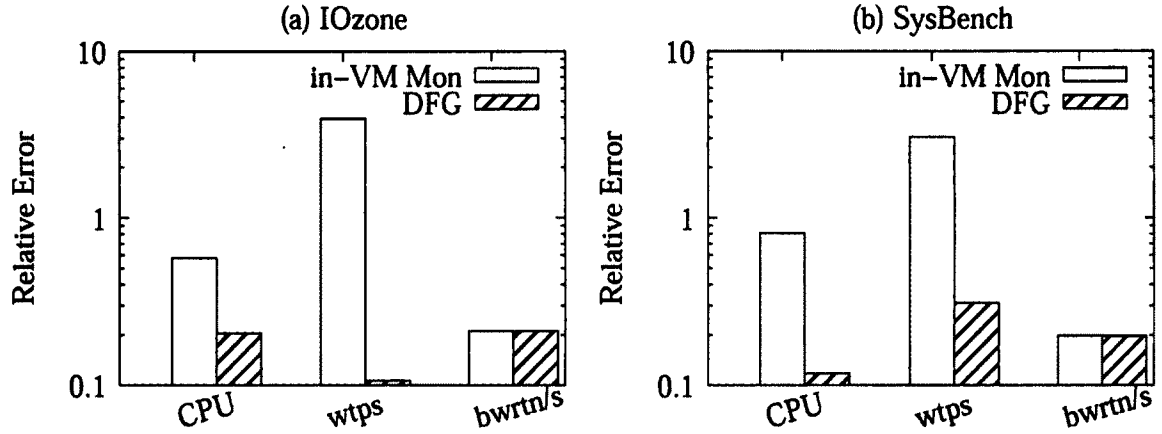


Figure 4.10: Relative error for in-VM monitoring and DFG model in mixed signal decomposing

problem and base our solution on a directed factor graph. We show how to build a base DFG model through benchmarking and design a run-time remodeling solution which is adaptive and guided by the base DFG model. Our evaluation shows that the proposed methodology is robust as it successfully calibrates the VM monitoring information and compares well to baseline measures.

5 Auto-Scaling of VMs in Resource Pools

Most modern hypervisors such as VMware ESX [20], Xen [34], and Microsoft Hyper-V [26] offer a rich set of resource control primitives at the individual virtual machine (VM) level to guide the sharing of physical resources among co-hosted VMs. For example, ESX offers the concepts of *reservations*, *limits*, and *shares* for both CPU and memory. Hyper-V provides *reserves*, *limits*, and *relative weights* for CPU, and *Startup RAM*, *Maximum RAM*, *buffer*, and *weight* for memory.

Additionally, VMware's Distributed Resource Scheduler (DRS) [69] offers the abstraction of a resource pool (RP), a logical container representing an aggregate resource allocation for a collection of virtual machines (VMs). One type of resource pool, referred to as a *virtual datacenter* (VDC), encapsulates an aggregation of resources allocated to an individual organization, either in a public or a private cloud. The configured capacity of the VDC can be tied to how much the organization is being charged. A VDC can contain multiple resource pools, each supporting a specific department of the organization; each resource pool can contain multiple applications, each supporting a specific business process.

Resource control settings can be specified at both a VM and an RP level. DRS [69] periodically divides the total capacity of a parent resource pool and distributes it to child VMs or RPs, according to the resource control settings and estimated re-

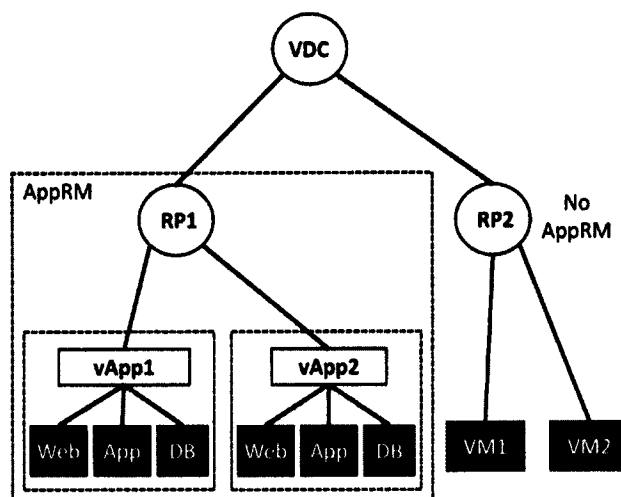


Figure 5.1: An example VDC containing two resource pools hosting two multi-tier vApps and two single-tier VMs.

source demands of individual VMs and RPs. This allows resources to flow across VMs or RPs in order to achieve better resource utilization.

Fig. 5.1 shows an example of a VDC containing two resource pools, RP1 and RP2. We refer to a virtual application running on one or more VMs as a *vApp*. RP1 contains two multi-tier vApps, each containing web, application, and database tier VMs, and running business critical production workloads. RP2 has two VMs running batch jobs that have less stringent performance requirements.

To protect the high priority applications from their neighbors, a VDC administrator can use the resource controls to achieve the following goals:

- *Resource guarantees*: Provide a guaranteed amount of a certain resource to a specific application or department within an organization, even when this resource is over-committed. For example, a resource **reservation** can be set on RP1 or on the individual VMs running important vApps.
- *Performance isolation*: Prevent demand spikes within one application or resource pool from affecting others. For example, a resource **limit** can be set on RP2 or its child VMs.

- *Proportional sharing*: Allow multiple applications or resource pools within the same organization to share resources in proportion to their relative priorities. For example, the administrator can set relatively higher **shares** on VMs running vApp1 than on those running vApp2 to provide performance differentiation between the two under resource contention.

Translating an application-level SLO to VM-level resource requirements is a well-known, difficult problem, due to the distributed nature of most modern applications, their dependency on multiple resource types, as well as the typically time-varying demands faced by the applications. There has been much research tackling this problem in the past several years [40, 102, 129], using statistical machine learning, fuzzy logic, as well as control theory. However, no prior work has studied automatic adjustment of resource control settings at the resource pool level. The industry has largely relied on heuristics [119], which is laborious and error-prone.

Therefore, even though resource pools offer additional powerful knobs to control resource allocation, converting application-level SLOs to these knob settings remains a challenging open problem. In this chapter, we propose a holistic solution that aims at providing SLO guarantees to individual applications within a resource pool hierarchy. The developed tool called AppRM is deployed on the VMware vSphere platform and is able to automatically translate application-level SLOs into individual VM or RP settings. To this end, AppRM employs a hierarchical architecture consisting of a set of *vApp Managers* and *RP Managers*. A vApp Manager determines the resource controls for a specific vApp and a RP Manager determines the resource controls for a specific resource pool.

Each vApp Manager contains a model builder, an application controller, and a resource controller. We use control theory and the online optimization approach in [102] as a building block for designing the model builder and the application controller within the vApp Manager. At the same time, we make the following key

contributions in the overall design of AppRM:

1. We design a **resource controller** in each vApp Manager that computes the desired resource control settings for the individual VMs. Although most prior work utilizes only *limits* or *shares*, AppRM specifically incorporates dynamic adjustment of resource *reservations* as an effective knob to ensure guaranteed access to specified amounts of resources.
2. We design an **RP Manager** that takes desired VM-level settings as inputs and computes the actual knob settings at *both* the VM and the RP levels, taking into account whether there is resource contention within the resource pool and whether the RP-level resource settings are *expandable* or *modifiable*. Furthermore, the RP Manager interacts with its associated vApp Managers asynchronously to relax timing constraints on the vApp Managers.

To the best of our knowledge, this is the first development of a holistic methodology that manages resource settings at both VM and resource pool levels. Our experimental results indicate that AppRM can achieve different targets for an application SLO in both under-provisioned and over-provisioned scenarios, in spite of dynamically-changing workloads. When the capacity of a VDC cannot satisfy the demands of all its applications, AppRM can ensure performance for the more important vApps by automatically adjusting the reservation and limit for the RP containing these vApps (e.g., RP1 in Fig. 5.1). If, however, the resource settings of this RP are unmodifiable, the RP Manager gracefully degrades the performance of all the child vApps in proportion to their respective demands.

5.1 Architecture

In Fig. 5.2, we show the overall architecture of the AppRM system that operates in the context of a single virtual application (vApp) to ensure that the vApp achieves

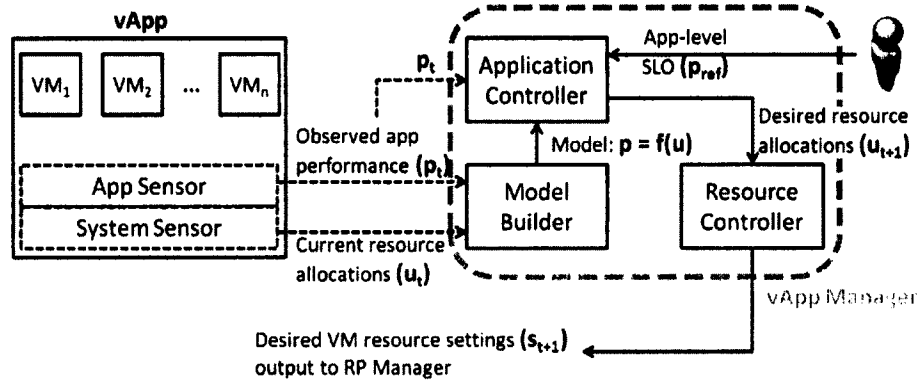


Figure 5.2: AppRM at work across VMs in a single vApp.

its user-defined, application-level SLO.

The App Sensor module collects various application-level performance metrics such as throughput and response times for each vApp. Note that an application may require more than one VM, e.g., a multi-tiered application. We use the System Sensor module to measure and keep track of current resource allocations for all the VMs associated with the target vApp. These two sets of statistics are input to the Model Builder module that first constructs and then iteratively refines a model for the observed application performance as a function of the VM-level resource allocations. The Application Controller module inverts this function to compute a new set of “desired” resource allocations in order to meet the user-defined application SLO. The Resource Controller module then determines a set of individual VM-level resource settings that would cause the VMs in the RP to acquire the desired resource allocations in the next control interval. Together, the Model Builder, Application Controller and Resource Controller modules constitute an instance of the vApp Manager for a single vApp.

In Fig. 5.3, we show how multiple vApps in a Resource Pool (RP) are managed by AppRM when they contend for resources in the RP. The App Manager from each vApp talks to the RP Manager where an Arbiter module addresses resource contention and computes the ideal values of VM-level and RP-level resource set-

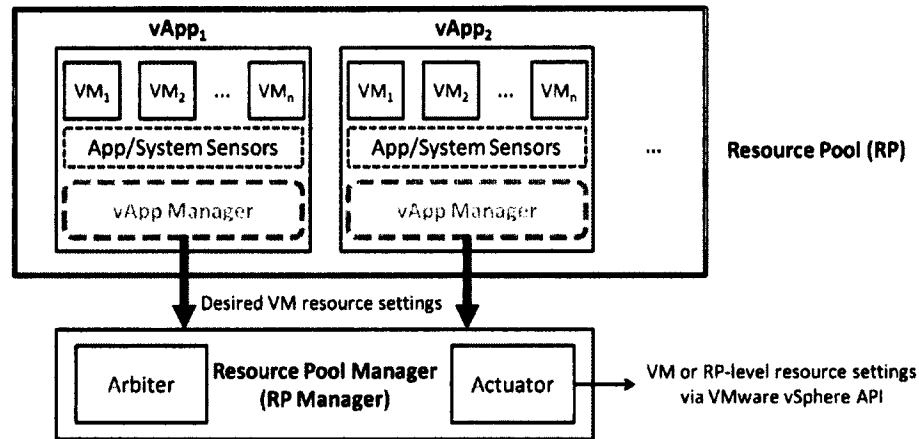


Figure 5.3: AppRM at work across vApps in a resource pool.

tings. These values are then set by the Actuator module, which uses the VMware vSphere WebServices API [24] to communicate with the inventory management layer.

5.2 Design

In this section, we describe the detailed design of each component module in AppRM. One set of sensor, model builder, application controller, and resource controller is instantiated for each application. There is one resource pool manager per resource pool.

5.2.1 Sensors

The sensor modules periodically collect two types of measurable statistics: real-time resource utilizations of individual VMs and application performance. Resource utilization statistics are collected by the system sensor through the vSphere Web Services API [24]. This API allows collection of an extensive set of ESX run time performance counters. We collect the average per-VM CPU utilization over a time interval using the `usage` performance counter, and the per-VM memory utilization using the `consumed` counter.

For application performance, we measure the request throughput, client side response time, and percentile response time. Although the application sensor is currently implemented in the client workload generator, it can potentially be implemented as a Hyperic [5] agent, which can provide performance statistics for a variety of applications without requiring any application modifications.

5.2.2 Model Builder

To determine the amount of resources needed for a vApp to meet its performance target, we first build a model of the relationship between the application resource allocation and its performance. As with most real-world systems, this relationship is often nonlinear and workload-dependent. Nevertheless, many nonlinear models can be approximated by a linear model in small-enough regions around an operating point. This linear model can then be updated periodically to adapt to changes in the workload and/or system conditions.

This is the fundamental assumption behind the online adaptive modeling approach in [102]. In this section, we explain how the same approach can be applied to our specific modeling problem. We first define, in Table 5.1, the key variables used in our model and the corresponding application controller.

For application $a \in A$, we define the resource allocation variable \mathbf{u}_a to be a vector that contains all measured resource allocations for application a . For example, for an application running in two VMs ($M_a = \{vm_1, vm_2\}$), if two resources are considered ($R = \{cpu, memory\}$), then \mathbf{u}_a is a vector $\mathbf{u}_a = (u_{a,vm1,cpu}, u_{a,vm1,mem}, u_{a,vm2,cpu}, u_{a,vm2,mem})$. $\mathbf{u}_a(t)$ represents the measured resource allocation values for application a during a control interval t .

In every control interval, the model builder recomputes the following auto-regressive-moving-average (ARMA) model that approximates the relationship between an ap-

Table 5.1: Notation

| | | | |
|---------------|---|--------------|---|
| A | set of applications | $u_{a,m,r}$ | measured allocation of resource type r in VM m of application a , |
| M_a | set of VMs in application $a \in A$ e.g., $M_a = \{vm_1, vm_2\}$ | | $0 \leq u_{a,m,r} \leq 1$ |
| t | index for control interval | p_a | measured performance of application a |
| R | set of resource types controlled e.g., $R = \{cpu, memory\}$ | p_a^{ref} | target performance of application a |
| $u_{a,m,r}^*$ | desired allocation of resource type r in VM m of application a , $0 \leq u_{a,m,r}^* \leq 1$ | \hat{p}_a | normalized performance of application a , where $\hat{p}_a = p_a/p_a^{ref}$ |
| | | p_a^{pred} | predicted performance of application a |
| | | \mathbf{b} | coefficient vector, $\mathbf{b} = [b_1 \dots b_n]^T$ |

plication's normalized performance and its resource allocations:

$$\hat{p}_a(t) = a(t)\hat{p}_a(t-1) + \mathbf{b}^T(t)\mathbf{u}_a(t). \quad (5.1)$$

Experiments here indeed confirm that this first-order ARMA model can predict application performance with satisfactory accuracy. The model is self-adaptive as its parameters $a(t)$ and $\mathbf{b}(t)$ are re-estimated in every control interval.

5.2.3 Application Controller

The application controller determines the resource requirements for all the VMs running the application such that the application can meet its performance SLO. Although we apply the same online optimization approach in [102] to the design of the application controller, we offer further discussions of the optimal controller solution and intuitive interpretation of the key parameters. More specifically, the controller seeks the VM-level resource allocation vector, $\mathbf{u}_a(t+1)$, for the next control interval $t+1$ that minimizes the following cost function:

$$J(\mathbf{u}_a(t+1)) = (p_a^{pred}(t+1) - 1)^2 + \lambda \|\mathbf{u}_a(t+1) - \mathbf{u}_a(t)\|^2. \quad (5.2)$$

Here, $p_a^{pred}(t+1)$ is the predicted value for the normalized application performance in interval $t+1$, using the model estimated in interval t (as shown in Eq. (5.1)), for

a certain resource allocation vector $\mathbf{u}_a(t+1)$. More specifically,

$$p_a^{pred}(t+1) = a(t)\hat{p}_a(t) + \mathbf{b}^T(t)\mathbf{u}_a(t+1). \quad (5.3)$$

The scaling factor, λ , captures the trade-off between the performance cost that penalizes the application's performance for deviating from its target (denoted by the normalized value equal to 1), and the stability cost that penalizes large oscillations in resource allocation values. Solving this quadratic optimization problem leads to the following optimal resource allocations:

$$\mathbf{u}_a^*(t+1) = (\mathbf{b}\mathbf{b}^T + \lambda\mathbf{I})^{-1} ((1 - a(t)\hat{p}_a(t))\mathbf{b} + \lambda\mathbf{u}_a(t)), \quad (5.4)$$

where \mathbf{I} is an identity matrix. Alternatively, let $u_{a,i}$ be the i -th resource allocation variable, then the above equation can be re-written as

$$u_{a,i}^*(t+1) = u_{a,i}(t) + \frac{1 - p_a^{pred}(t+1)}{\lambda + \sum_{i=1}^n b_i^2} b_i. \quad (5.5)$$

We make the following key observations:

(1) When $b_i = 0$, indicating no impact from the i -th resource allocation variable on the application performance, then the i -th resource allocation variable will see no change in the next control interval;

(2) When $b_i > 0$, indicating a positive correlation between the i -th resource allocation variable and the performance value, if the model-predicted performance is below the target, i.e., $p_a^{pred}(t+1) < 1$, then the i -th resource allocation variable will be increased such that the performance value can be increased in the next interval; and the opposite is true if $b_i < 0$ or if $p_a^{pred}(t+1) > 1$.

(3) The scaling factor λ affects the amount of resource allocation changes. As λ increases, the oscillation in each resource allocation variable is reduced.

5.2.4 Resource Controller

The goal of the resource controller is to translate the optimal resource allocation values computed by the application controller to desired VM-level resource control settings. The translation is needed for two reasons:

- The output of the application controller is in percentage units, whereas the reservation and limit values for both CPU and memory are in absolute units, i.e., megahertz (MHz) or megabytes (MB).
- We explicitly allocate more resources than the computed values as a "safety buffer", to deal with inaccuracies in the computed optimal allocations.

The pseudo-code in Algorithm 5.1 summarizes the algorithm applied to every VM within the same vApp, for both CPU and memory resources. The algorithm calculates the resource capacity based on the specific resource type (line 4-7). The desired resource reservation is computed by multiplying the optimal value and the capacity (line 8). The "safety buffer" size is determined by the reservation, the normalized performance, and a precomputed constant value *delta* (line 12). In this algorithm, we set *delta* to a *low* or *high* value depending on whether the measured application performance is below or above the target (line 10). When the performance is better than the SLO ($perf < 1$), a relatively small buffer size can reduce the performance fluctuation around its target. When the performance is worse than the SLO ($perf > 1$), a relatively large buffer size is needed to improve the performance convergence rate. We set $low = 0.1$ and $high = 0.3$ after experimenting with different values. The resource limit value is set to the sum of the reservation and buffer size (line 12). We also explored additional ways of setting the *limit* value including: 1) $limit \leftarrow resv$, 2) $limit \leftarrow resv + const$, and 3) $limit \leftarrow resv + perf * resv$. The one presented in the algorithm has the best performance. The nonzero, adaptive buffer between limit and reservation allows the resource scheduler to adjust

run time allocations if needed. The limit is then compared against the available capacity and the minimum capacity a VM needs in order to ensure that the final value is feasible (line 13-17).

Algorithm 5.1: Calculate desired Reservation and Limit settings

Input : optimal allocation u^* , resource type $type$, and normalized performance $perf$ (i.e., \hat{p}_a)
output: Reservation and Limit value pair

```

1 if  $u^* < 0$  then  $u^* \leftarrow 0$ ;
2 if  $u^* > 1$  then  $u^* \leftarrow 1$ ;
3  $capacity \leftarrow 0$ ;
4 if  $type = CPU$  then
5   |  $capacity \leftarrow \text{getNumVirtualCPUs}() * \text{getCPUMHz}()$ ;
6 else if  $type = MEM$  then
7   |  $capacity \leftarrow \text{getMemoryMB}()$ ;
8  $resv \leftarrow u^* * capacity$ ;  $delta \leftarrow 0$ ;
9 if  $perf < 1$  then  $delta \leftarrow low$ ;
10 else  $delta \leftarrow high$ ;
11  $buffer \leftarrow delta * perf * resv$ ;
12  $limit \leftarrow resv + buffer$ ;
13 if  $limit > capacity$  then  $limit \leftarrow capacity$ ;
14 if  $type = CPU$  then
15   |  $limit \leftarrow \max(MINCPU, limit)$ ;
16 else if  $type = MEM$  then
17   |  $limit \leftarrow \max(MINMEM, limit)$ ;
18 return  $\langle resv, limit \rangle$ ;
```

5.2.5 Resource Pool (RP) Manager

For each resource pool, an RP Manager determines the allocation of resources to the applications running under this RP, based on the resources requested by the associated vApp Managers, the available resource pool capacity, and resource pool settings. This is required because the vApp Managers act independently of one another and may, in aggregate, request more resources than the resource pool has available.

A resource pool is defined as *modifiable* if the RP-level resource settings can be

modified by an administrator or through an API. This allows the resources to flow from low priority RPs to high priority RPs (e.g., from RP2 to RP1 in Fig. 5.1) within the same VDC. In addition, a resource pool is *expandable* if it can automatically increase its own reservation, when it is exceeded by the sum of its children's reservations. Suppose that a resource pool has a reservation R , and VM_i represents a virtual machines under this RP with a reservation r_i . Then we have the following constraints:

| Constraints | |
|----------------|-------------------|
| Expandable | $\forall r_i < R$ |
| Non-expandable | $\sum_i r_i < R$ |

For a resource where the total reservation requested is less than the RP-level reservation, the RP Manager honors each vApp Manager's requests. For a contested resource where the sum of VM-level reservations is greater than the RP-level reservation, the following attribute information about the resource pool has to be considered:

1. Expandable: The RP Manager sets the VM-level resource knobs directly as requested by the vApp Managers, and the RP expands its reservation automatically to accommodate the total demand.
2. Non-expandable but modifiable: The RP Manager modifies the RP-level reservation to the sum of the requested VM-level reservations.
3. Non-expandable and unmodifiable: The RP Manager throttles the VM-level reservations in proportion to the requested reservation for each VM.

Each vApp Manager sends the resource setting requests to the RP Manager periodically. Each request is a 4-tuple of VM name, resource type, requested reservation and requested limit: $(vmname, type, resv, limit)$. For each resource, the RP Manager maintains a global vector (e.g., `cpu_alloc_table`). Each element in the

vector is a 3-tuple of VM name, requested reservation, and allocated reservation: $(vmname, req_resv, alloc_resv)$. This table is updated to keep track of each VM's resource requests over time.

The RP Manager processes the requests from vApp Managers asynchronously. Instead of waiting for all managed vApps to send their requests and processing them all, the RP manager is designed to start processing the vApp request and then setting the values when a request arrives. Once receiving a request from a vApp Manager, the RP Manager runs Algorithm 5.2 to calculate the actual VM-level and RP-level resource values. Based on the attribute of the resource pool, the values are set by the Actuator differently. This design allows the vApp Managers under the same RP Manager to have different control intervals based on different application needs.

Algorithm 5.2 first checks the expandability and modifiability of the resource defined by *type* (line 1-2). It reads the currently used and available RP reservation settings and computes the total RP capacity (line 3-5). Then it gets the requesting VM's current reservation value (line 6). If the RP is expandable or modifiable, it sets the VM reservation value directly (line 8). If the RP is non-expandable and unmodifiable, the RP Manager scans the global vector to get the total value of the requested reservations from all VMs in the resource pool and computes the proportionally-allocated value *prop_resv* for the requesting VM (line 10-12). The RP-level reservation and limit are set and the global vector is updated with the new tuple $(vname, resv, vm_resv)$ in the end (line 15-17).

Depending on the expandable and modifiable attribute of the resource pool, the RP Manager sets the calculated values as follows:

1. Expandable: only VM-level setting (vm_resv, vm_limit) changes are applied while RP-level settings are handled automatically by VMware resource management system.

Algorithm 5.2: Calculate VM and RP settings

Input : vApp Manager request tuple (*vmname*, *type*, *resv*, *limit*);
Output: VM and RP level setting tuple
(*vm_resv*, *vm_limit*, *rp_resv*, *rp_limit*)

```
1 isExpandable  $\leftarrow$  isExpandable (type);
2 isModifiable  $\leftarrow$  isModifiable (type);
3 curRpResv  $\leftarrow$  getRpReservationUsed (type);
4 rpAvail  $\leftarrow$  getRpAvailReservation (type);
5 rpCapacity  $\leftarrow$  curRpResv + rpAvail;
6 curVmResv  $\leftarrow$  getVmReservation (vmname, type);
7 if isExpandable or isModifiable then /* Expandable or modifiable */
8 |   if resv > rpCapacity then vm_resv  $\leftarrow$  rpCapacity;
9 |   else vm_resv  $\leftarrow$  resv;
   else /* Non-expandable and unmodifiable */
10 |   others_resv_req  $\leftarrow$  totalResvReqExceptVM (vmname);
11 |   total_rp_resv_req  $\leftarrow$  others_resv_req + resv;
12 |   prop_resv  $\leftarrow$  rpCapacity * resv / total_rp_resv_req;
13 |   vm_resv  $\leftarrow$  min(prop_resv, rpAvail + curVmResv, resv);
14 if limit < vm_resv then vm_limit  $\leftarrow$  vm_resv;
15 else vm_limit  $\leftarrow$  limit;
16 rp_resv  $\leftarrow$  vm_resv + curRpResv - curVmResv;
17 rp_limit  $\leftarrow$  rp_resv;
18 updateTable (vmname, type, resv, vm_resv);
19 return < vm_resv, vm_limit, rp_resv, rp_limit >
```

2. Non-expandable but modifiable: Case 1: apply both VM-level and RP-level changes; Case 2: only RP-level reservation and limit (*rp_resv*, *rp_limit*) are set. For case 2, VM level settings are not applied explicitly, but rely on VM *shares* to adjust resource allocations among resource-competing VMs.
3. Non-expandable and unmodifiable: only proportionally-throttled VM-level values (*vm_resv*, *vm_limit*) are set.

The actuator module sets the resource reservation and limit values of virtual machines through the vSphere Web Service API [24]. The resource (CPU or memory) reconfiguration of a virtual machine is done by one explicit call of the `reconfigVM_task` method.

5.3 Testbed Setup

For testing the effectiveness of *AppRM*, we chose MongoDB [11] as the benchmark application because it is fairly representative of a modern distributed data processing application. As shown in Fig. 5.4a, we set up a MongoDB (version 1.8.1 Linux 64-bit) cluster consisting of 3 VMs. VM-1 and VM-2 are MongoDB shards running *mongod* instances. VM3 runs a *mongos* instance, which load balances and routes queries to the shards. A configuration *mongod* server that stores the metadata for the MongoDB cluster also runs on VM-3. The MongoDB application defines two types of transactions, each of which can be generally classified as a “read” or “write” operation.

For generating dynamically changing workloads, we chose Rain [37] as our workload generation toolkit. Rain provides the ability to generate variable amounts of load in multiple patterns along with different mix of operations. Here we assume that the workload is defined by two characteristics: the number of clients and the percentage of read/write operations.

Each of the VMs in the MongoDB cluster have 2 vCPUs while the Rain VM has 2 vCPUs. All of these VMs have been configured with memory size of 4GB. We run the Rain VM and the MongoDB cluster VMs on two separate ESXi 5.0 virtualized hosts. We also run some co-hosted VMs on ESX2 along with the MongoDB cluster VMs to induce some resource contention. The full host configuration is shown in Table 5.2. All VMs run Linux Ubuntu 2.6.35 as their guest operating system. *AppRM* runs on a separate VM on ESX1 along with Rain.

Table 5.2: Configuration of hosts

| Host | ESX1 | ESX2 |
|---------|--|---|
| Model | HP ProLiant BL460c G7 | HP ProLiant BL465 G7 |
| CPU | Intel Xeon CPU X5650 24 cores @ 2.10GHz | AMD Opteron 6172 12 cores @ 2.67 GHz |
| Memory | 128 GB | 96 GB |
| Storage | DGC Fibre Channel Disk | DGC Fibre Channel Disk |

5.4 Performance Evaluation

In this section, we present our experimental results for AppRM. These experiments are designed to test the following capabilities:

1. Enforce performance targets for metrics including mean response time, throughput, and 95th response time percentile.
2. Automatically detect and mitigate dynamically-changing workload demands
3. Apply control on multiple applications
4. Enforce performance targets under competing workloads

5.4.1 Achieving Performance Targets for Multiple Metrics

This scenario is designed to evaluate the effectiveness of AppRM with different performance targets. For this set of experiments, we use the setup shown in Figure 5.4(a), where two physical nodes host the three MongoDB application VMs, workload generator VM, and *AppRM* VM. This set of experiments allow us to gain insight into the system behavior and validate the internal working of the model builder and controller.

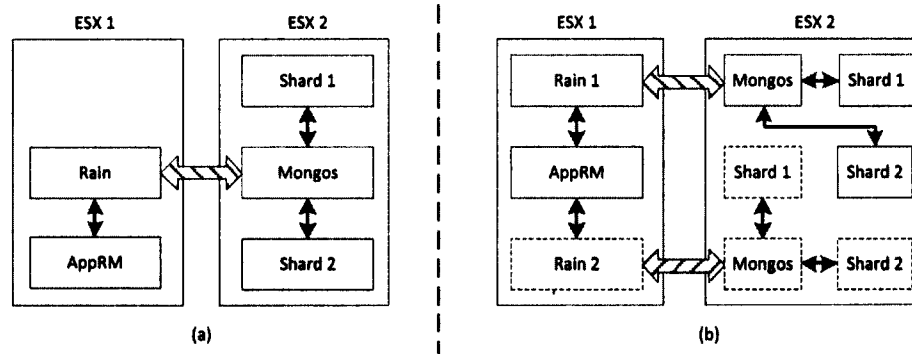


Figure 5.4: Experimental setup with a MongoDB cluster and Rain benchmark

In this experiment, we set the MongoDB application with 300 threads emulating 300 concurrent clients connecting to the MongoDB server. The workload is composed of 50% read and 50% write requests. For each emulated client, there is no thinking time between receiving the last reply and sending the next request. We set the target for mean response time as 300 milliseconds. Figure 5.5 shows how performance changes across time under two initial resource settings: under-provisioning¹ and over-provisioning². The under-provisioning describes a scenario where the initial settings of the application VM resources are not sufficient to meet application needs leading to high response time. In the Over-provisioning case, initial VM resource settings are more than the application needs leading to resource over-allocation and wastage. Figure 5.5 illustrates the performance for both cases

¹all VMs are set to $R_{cpu} = R_{mem} = 0$, $L_{cpu} = L_{mem} = 512$ (MHz/MB)

²all VMs are set to $R_{cpu} = R_{mem} = 0$, $L_{cpu} = L_{mem} = \text{unlimit}$

as a function of the control intervals. Note that, "white" area represents the period of model learning, where only the sensor module is active. The "gray" area represents the period in which all modules are activated, during which the system model is updated and the control actions are performed. We can see that AppRM can adjust the resource allocations to achieve the target in both under-provisioned and over-provisioned cases.

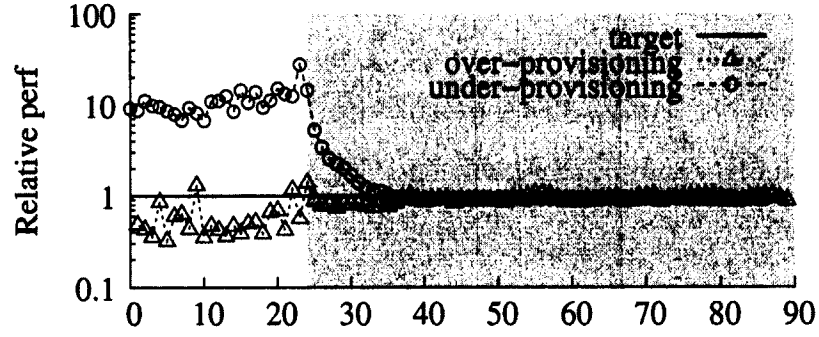


Figure 5.5: Mean response time target (300 ms)

Figure 5.6 shows the resource utilization changes for all the MongoDB VMs with under-provisioning initial settings. It is apparent that for both CPU and memory resources, the initial allocation is not sufficient for the application to achieve its target. When AppRM is turned on, the resource allocation automatically increases to meet the target.

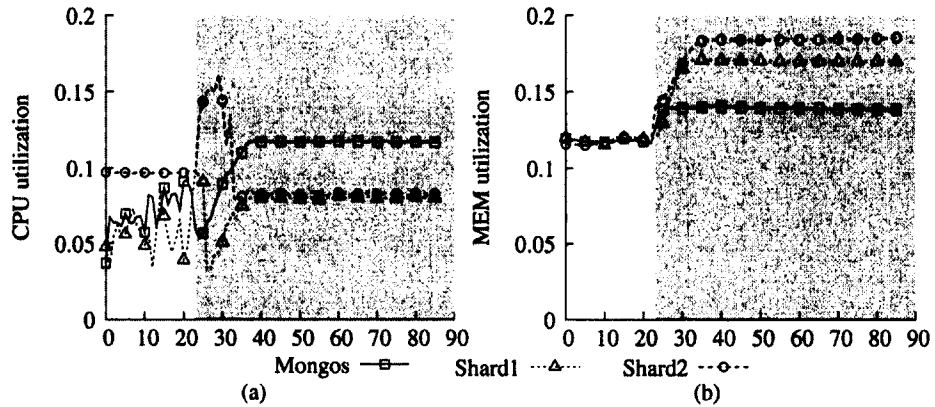


Figure 5.6: Resource utilization for under-provisioning

Naturally, if the amount of allocated resources is more than the target, then

it maybe preferable to reduce its allocation. This effect is shown in Figure 5.7, especially for CPU. The utilization of CPU is reduced and stabilized in Figure 5.7(a) after AppRM is turned on. Figure 5.7(b) shows that memory is hardly affected in this case. This is because of the design of the Application controller which aims to meet the target with minimal resource changes.

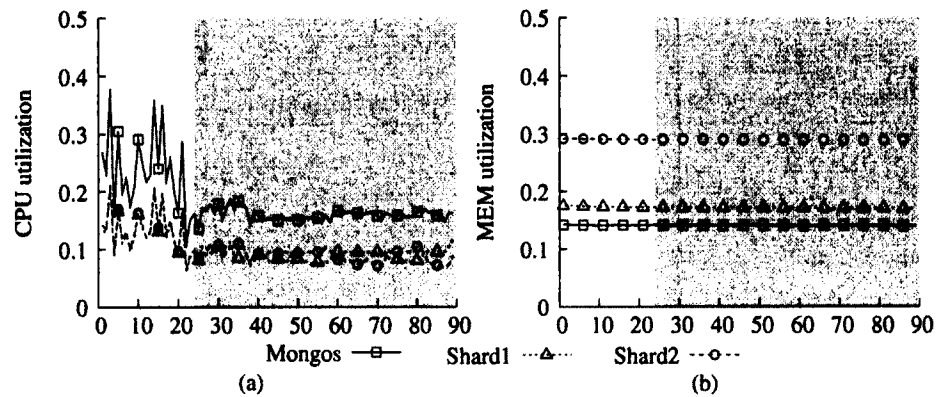


Figure 5.7: Resource utilization for over-provisioning

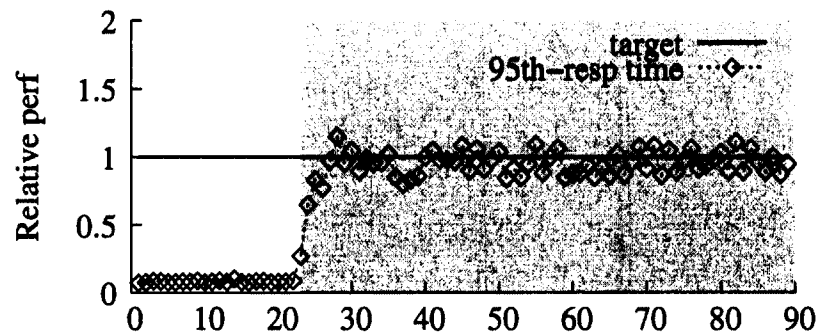


Figure 5.8: 95th percentile response time target (2000 ms)

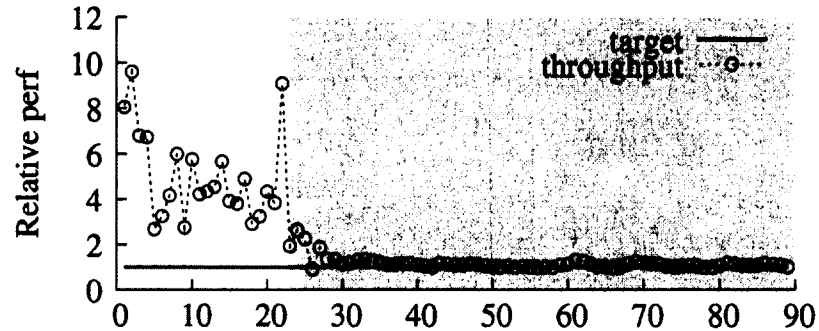


Figure 5.9: Throughput target (50,000 reqs/s)

Can AppRM work correctly with other performance targets, especially percentile

Table 5.3: Definition of three changing workloads

| Target mean RT | Period 1 | | | Period 2 | | Control interval |
|-------------------|----------|----------------|---|----------|----------------|------------------|
| | clients | read/write mix | | clients | read/write mix | |
| 500ms | 300 | 50r/50w | → | 500 | 50r/50w | 1 min. |
| 600ms | 300 | 50r/50w | → | 500 | 80r/20w | 1 min. |
| 800ms | 300 | 50r/50w | → | 500 | 20r/80w | 5 min. |

response time, which is intrinsically difficult to control? Figures 5.8 and 5.9 answer this question. They show the experiment results with the same experiment setup but different performance targets, i.e., 95th-percentile response time and throughput. AppRM successfully adjusts the resources settings to meet these targets as seen in the figures. In summary, results in scenario 1 show that for different performance targets, AppRM can allocate the right amount of resources for each application VM to meet end-to-end SLOs.

5.4.2 Detecting and Mitigating Dynamically-Changing Workload Demands

In this scenario, we evaluate the effectiveness of AppRM in meeting the target SLO under dynamically-changing workloads. Table 5.3 defines the three experiments with changing workload conditions. In all three experiments, a mean response time target is used. Here, we intentionally show different target values across experiments to demonstrate AppRM's robustness within a dynamic environment. Additionally, in the third experiment, we change the control interval from 1 minute to 5 minutes to demonstrate that AppRM also works correctly for different intervals.

In the first experiment, we use a target mean response time of 500 milliseconds. The experiment starts in an over-provisioned condition ($R_{cpu} = R_{mem} = 0$, $L_{cpu} = L_{mem} = unlimited$) before AppRM is turned on. Fig. 5.10 shows that during Period 1 (interval 0-90), the normalized mean response time gradually converges to the performance target in 15 minutes. During Period 2 (91-150), the workload increases

to 500 clients as indicated in Table 5.3. The normalized performance deteriorates to 1.41 but quickly converges back to its target.

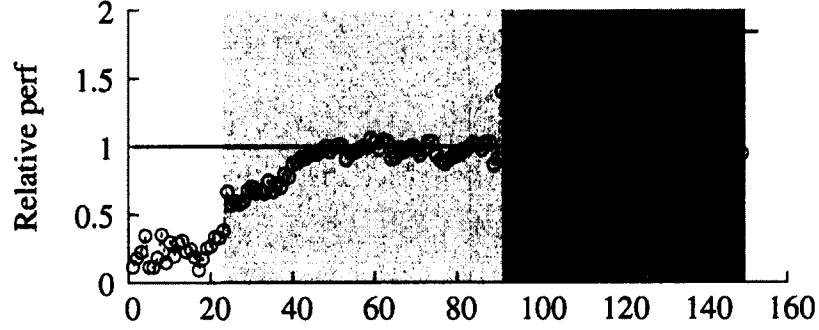


Figure 5.10: Measured performance under dynamic workloads with target 500 ms

In the second experiment, the target mean response time is set to 600 milliseconds. Fig. 5.11 shows that the application is over-provisioned initially ($R_{cpu} = R_{mem} = 0$, $L_{cpu} = L_{mem} = \text{Unlimited}$) resulting in very low response times. AppRM gradually reduces its resources allocations (see Fig. 5.12) while maintaining the mean response time near its target. At interval 61, the workload increases to 500 clients and becomes read-intensive (with 80% of read operations). This change does not cause a significant increase of response times and thus all VMs' resource utilizations are relatively stable, as shown in Fig. 5.12.

In the third experiment, the target mean response time is set to 800 ms. Fig. 5.13 shows that in Period 1 (0-90), the application performance is initially up to 10 times worse than its target, due to the initial under-provisioned resource settings ($R_{cpu} = R_{mem} = 0$, $L_{cpu} = L_{mem} = 512$ (MHz/MB)). At interval 23, AppRM is activated and it increases both CPU and memory allocations (see Fig. 5.13), bringing the measured performance down to its target. In Period 2 (91-150), 200 more threads are added to the workload client and the workload is dominated by write operations (80% of operations are writes). This sudden workload increase initially degrades the application performance, up to 50% worse than the target (see Fig. 5.13). AppRM rapidly responds to this change and correctly re-adjusts the resource allocations.

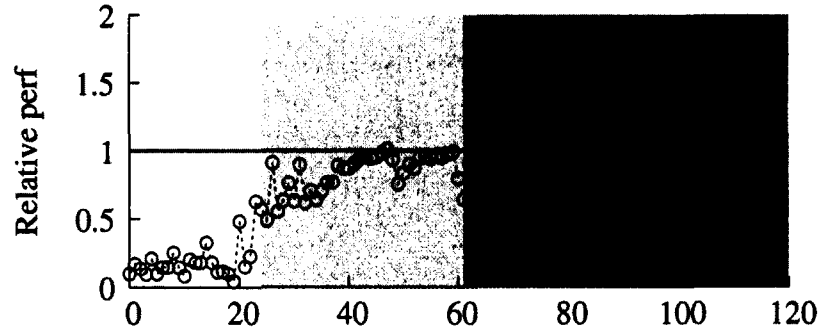


Figure 5.11: Measured performance under dynamic workloads with target 600 ms

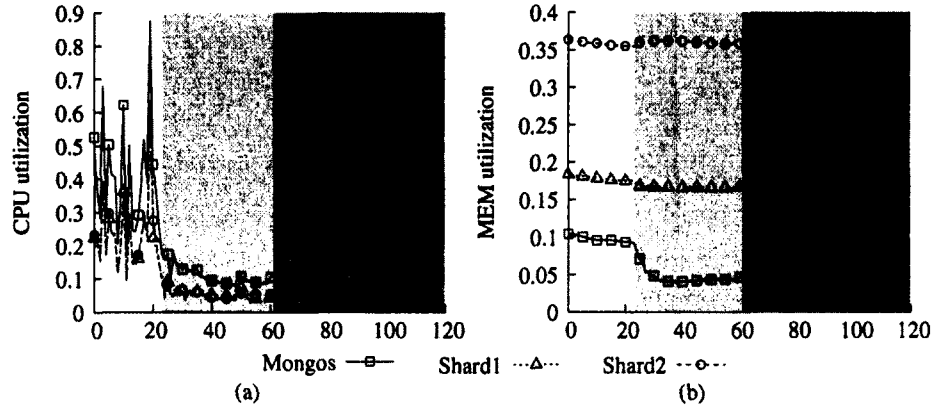


Figure 5.12: Resource utilization for dynamic workloads

Note that both CPU and memory utilizations of the MongoDB VMs are increased (see Fig. 5.14), to mitigate the increased workload while maintaining the target response time.

5.4.3 Applying Control on Multiple Applications

In this scenario, we evaluate the case of multiple AppRM instances to control multiple applications. This experiment has the setup shown in Figure 5.4(b), where two physical nodes host two sets of MongoDB applications, workload generators, and AppRMs. When there is no resource contention, the two AppRMs work independently in monitoring and adjusting application resources. The benefit of having two instances of AppRMs simplify the design of AppRM and allow users setting different performance metrics and target at will.

Figure 5.15 shows the achieved mean response time with different targets: app1

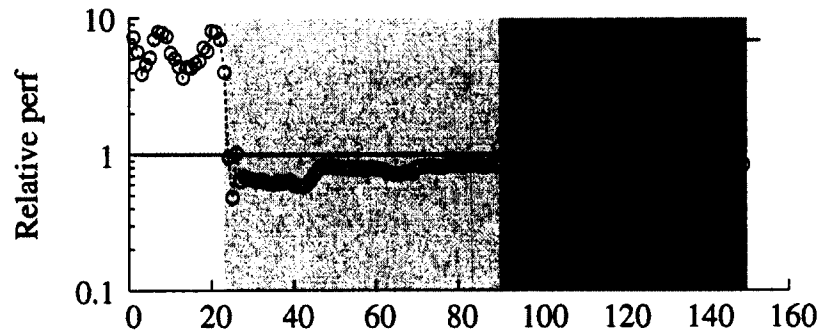


Figure 5.13: Measured performance under dynamic workloads with target 800 ms

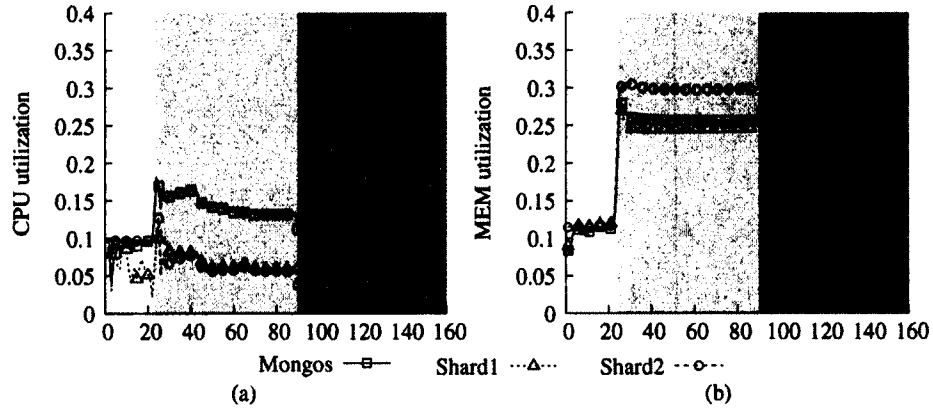


Figure 5.14: Resource utilization for dynamic workloads

has mean response time of 1000 millisecond as target and app2 has mean response time of 600 millisecond. It is clear that AppRM helps for performance to converge to the target line after it is activated at interval 23. Note that in this scenario, two AppRM instances do not have resources contention, which means that the aggregate resource demands is less than the configured resource pool size or host size. Several cases of contention are discussed in the next section.

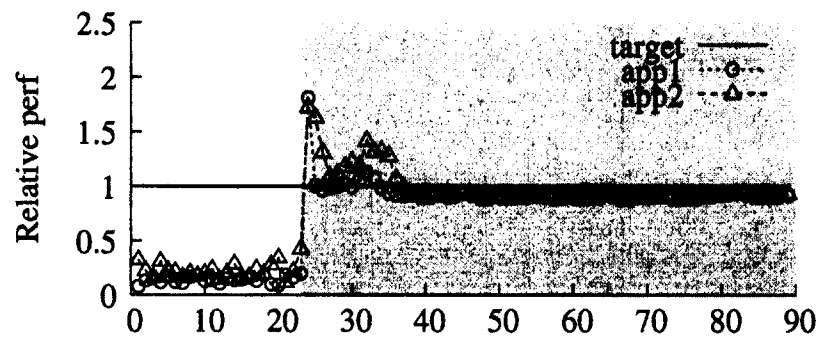


Figure 5.15: Application 1 mean target 1000 ms and Application 2 target 600 ms

5.4.4 Enforcing Performance Targets under Competing Workloads

The testbed setup for the experiments in this scenario is shown in Figure 5.16. Two instances of MongoDB application are configured within a resource pool in order to have a better isolation of all resources in a cluster. We run 10 VMs outside the resource pool in the same cluster/host to mimic intensive competing workloads. Each competing VM is running a CPU intensive job to consume all possible CPU resources once allocated. We show that AppRM is able to help the applications to achieve performance targets under different resource pool settings.

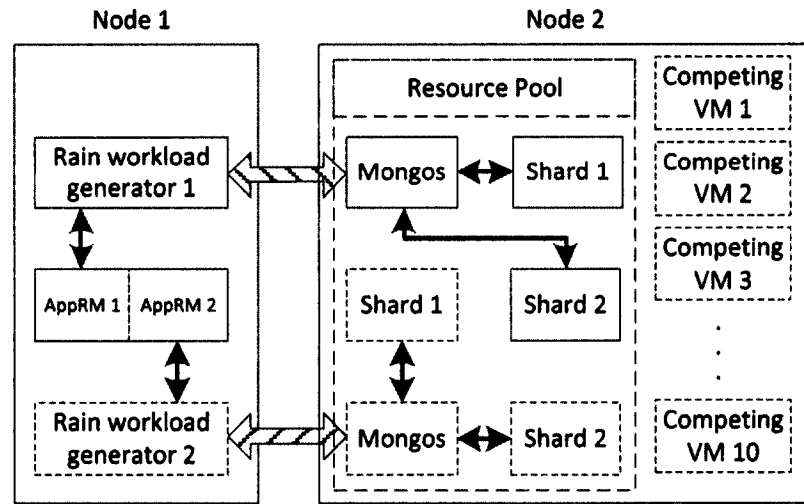


Figure 5.16: Testbed setup for resource pool experiments

5.4.4.1 Non-expandable and Unmodifiable Resource Pool

In this scenario, CPU of the resource pool is configured to be non-expandable and the resource pool settings are unmodifiable at run time. If the resource pool is large enough and can always satisfy the non-expandable constraints, then every VM requested resource is always satisfied and the result is the same as is Section 5.4.3. If the resource pool is insufficient for satisfying reservation of all virtual machines combined, then AppRM has to throttle reservation values back proportionally to

the requested reservation value. Otherwise, the resource setting operation of each VM could fail due to invalid value. We show a case result with initial settings as following:

| | Reservation | Limit |
|---------------|-------------|----------|
| Each VM | 0 | unlimit |
| Resource Pool | 2000 MHz | 2000 MHz |

This setting makes a clear partition of CPU resources between resource pool and the rest 10 CPU intensive competing VM. Both application have a mean response time equal to 600ms as targets. Figure 5.17 shows that AppRM can not help to improve the performance because resource pool itself does not have sufficient resource and not allowed to increase its size. AppRM throttles back each VM requested reservation settings proportionally and balances the performance degradation of the two applications.

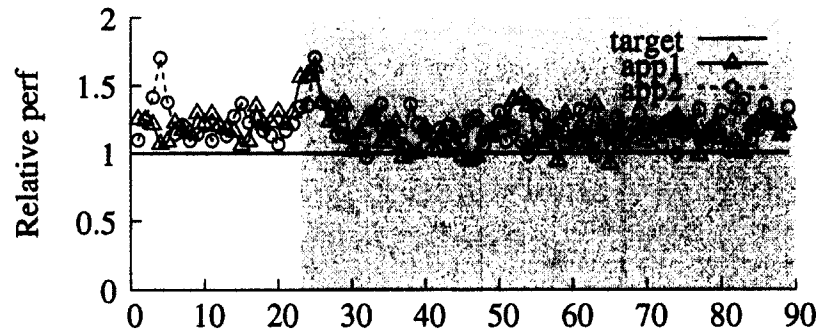


Figure 5.17: Measured application relative performance in non-expandable and unmodifiable RP (targets: 600ms)

5.4.4.2 Non-expandable but Modifiable Resource Pool

In this scenario, CPU of resource pool is non-expandable but resource pool reservation value is allowed to change at run time. There are three cases based on the methodologies for setting VM and RP values:

Case 1: AppRM change both RP and VM resource settings according to Algorithm 5.2. The benefit of this method is to allow AppRM take fine-grained control

on the per-VM resource setting and needs to be set accurately. The experiment for this case has the initial setting:

| | Reservation | Limit | Share |
|---------------|-------------|---------|--------|
| Each VM | 0 | unlimit | Normal |
| Resource Pool | 0 | unlimit | Normal |

Figure 5.18 illustrates the measured performance as a function of control intervals. The initial setting does not provide any guaranteed CPU resource for both applications but let applications compete CPU resources with the 10 CPU intensive VMs. The allocated CPU resource based on shares is insufficient for both applications to meet their performance targets as shown in the first 23 intervals. When their AppRM controllers are activated, they change both RP and VM reservation values and help both applications meet their targets. Figure 5.19 shows how the application level and RP reservation change over time.

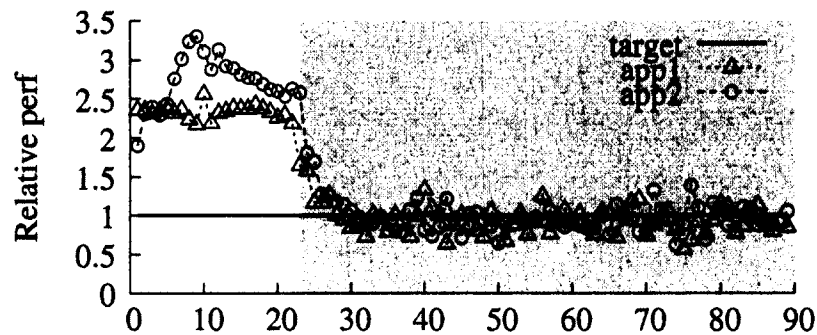


Figure 5.18: Measured application relative performance in non-expandable and modifiable RP (targets 600ms)

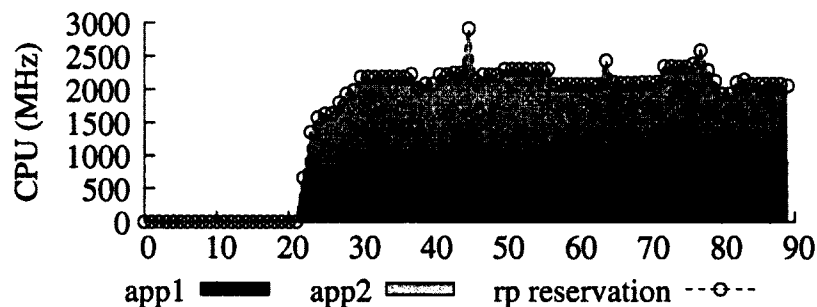


Figure 5.19: Application and RP level reservations

Case 2: In this case, AppRM only changes the RP reservation setting. For individual VMs within the resource pool, they are always set to $R = 0$, and $L = \text{unlimit}$. The benefit of this method is its easy setting. When resources in the RP are contested, it relies on shares to regulate resource allocation. The initial setting is shown in the table:

| | Reservation | Limit | Share |
|---------------|-------------|---------|--------|
| Each VM | 0 | unlimit | Normal |
| Resource Pool | 0 | unlimit | Normal |

It is apparent that during the initial (interval 0-23), the inferior performance in Figure 5.20 is improved by the increased resource pool reservation value (see Figure 5.21) as set by AppRM. Although individual VMs are not set with specific reservations, the RP level reservation setting guarantees the right amount of CPU resource which can be shared with application VMs.

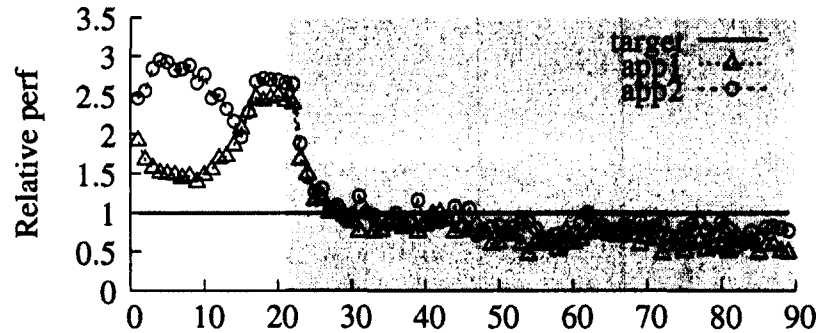


Figure 5.20: Measured application relative performance in non-expandable and modifiable RP (targets 600ms)

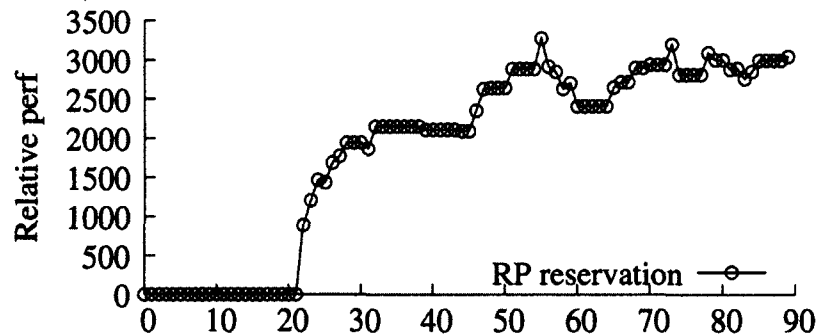


Figure 5.21: RP level reservation

Case 3: This case differs from case 2 in only one point, i.e. the RP limit value is to

set the same as the reservation value. This method makes a clear CPU resource isolation between RP and the 10 CPU intensive VMs. Still, for individual VMs in the resource pool, these values are always set to $R = 0$, and $L = \text{unlimit}$. AppRM never sets the reservation and limit values on individual VMs. The initial settings are as follows:

| | Reservation | Limit | Shares |
|---------------|-------------|----------|--------|
| Each VM | 0 | Unlimit | Normal |
| Resource Pool | 2000 MHz | 2000 MHz | Normal |

Figure 5.22 demonstrates that the performance of both applications improves after their AppRM controllers are activated. The reservation and limit value of RP increase to a proper value which can guarantee the needed CPU resource of two applications, see Figure 5.23.

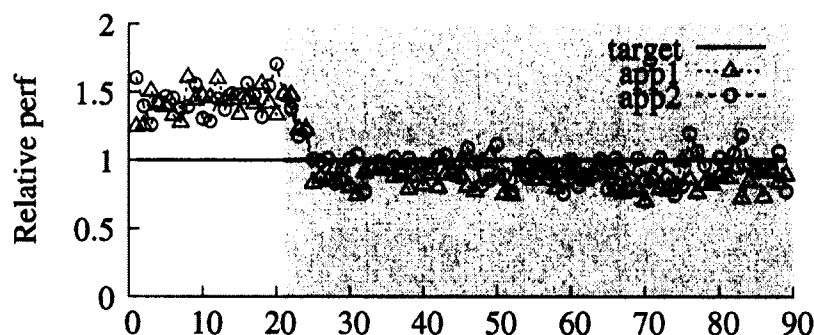


Figure 5.22: Measured application relative performance in non-expandable and modifiable RP (targets 600ms)

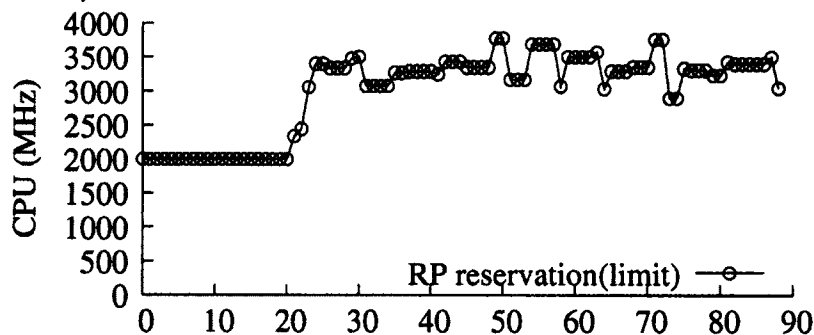


Figure 5.23: RP level reservation setting

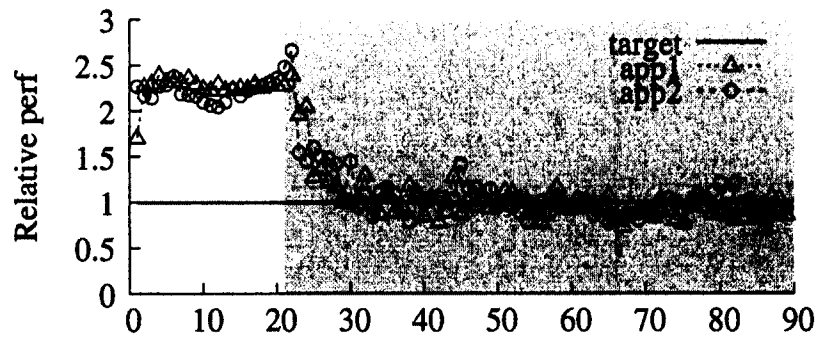


Figure 5.24: Measured application relative performance in expandable RP (targets 600ms)

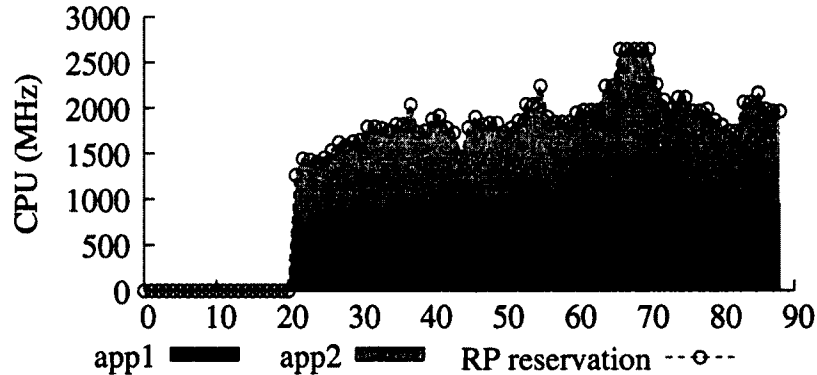


Figure 5.25: Application and RP level reservations

5.4.4.3 Expandable Resource Pool

Expandable reservation allows the resource pool to expand automatically when the combined reservation of all virtual machines is larger than the resource pool reservation. Under this situation, AppRM only needs to check the Expandable Constraints and set the VM level settings. We use the following initial settings for their experiment:

| | Reservation | Limit | Shares |
|---------------|-------------|---------|--------|
| Each VM | 0 | Unlimit | Normal |
| Resource Pool | 0 | Unlimit | Normal |

Figure 5.24 shows that the allocated CPU resource based on share is not enough to allow the applications to meet their targets. When the AppRM controllers are activated, the guaranteed reservation of CPU to each application VM is set, thus it allows the applications to meet their targets. Figure 5.25 shows the increase of the

two application level reservations.

5.5 Summary

This chapter gave an overview on *AppRM*, a performance management tool that automatically adjusts resource control settings at individual virtual machine level or at resource pool level such that the virtualized applications running in a virtual datacenter can meet their respective performance goals. The experimental results demonstrate the effectiveness of *AppRM* in a number of different scenarios, with one or more virtualized applications either under dynamically changing workloads, or under resource contention among neighboring applications within the same resource pool or within the same virtual data center.

6 Predictive VM Consolidation

To shield application performance from infrastructure management and performance interference due to VM co-location, resource management in virtualized data centers requires careful VM placement that avoids (or minimizes) resource contention on diverse physical resources in order to control queueing delays. From the management perspective, assuming that all applications have the same priority, resources should be allocated to the application VMs in an egalitarian manner unless specific, per application QoS targets need to be met. With multiple VMs exhibiting diverse resource demands, it is natural to assume that these VMs should be placed on servers in a way that server resources are best utilized, such that single bottlenecks are minimized.

Originally introduced in network management, fair load balancing ensures fair resource allocation among users through balancing of the server workload. This can lead to desirable system properties (e.g., Pareto efficiency [64]) in data center resource management. While previous work on fair load balancing focuses on a single resource (e.g., network), in this chapter, we consider a more complicated and challenging problem: how to best match *multiple* resources with specific VMs. The new methodology presented here is called *predictive virtual machine consolidation* (PREMATCH) and focuses on how to best co-locate different VMs within different servers such that performance interference between co-located VMs is minimized. PREMATCH targets max-min fairness in VM resource allocation and min-max load

balancing in server workload assignment.

Load normalization and total ordering are two entwined research challenges in multi-resource load balancing. In single-resource load balancing, candidate solutions can always have a total order, while in multi-resource load balancing, there may *not* be a single solution that is best across each resource dimension. For example, consider a two-machine system, two resource dimensions (CPU and disk), and two candidate consolidations: one consolidation defines the load vectors as $A_1 = \langle 0.9(cpu), 0.7(disk) \rangle$, $A_2 = \langle 0.5(cpu), 0.7(disk) \rangle$; another proposes the load vectors as $A_1 = \langle 0.7(cpu), 0.6(disk) \rangle$, $A_2 = \langle 0.7(cpu), 0.8(disk) \rangle$. From these two consolidations, the maximum load on the first one is on the CPU, while the maximum on the second one is on the disk. It is completely unclear to determine without a priori experimentation which of the two consolidations reduces the execution time of *both* applications and why. The problem is further exacerbated by introducing more resource dimensions and more servers as performance interference between applications and different co-location schemes may vary dramatically. Exhaustive experimentation to select the best of all possible combinations is non-feasible, especially within a dynamic environment.

The focus of this chapter is on the development of a robust *prediction framework* that suggests how to best match multiple resources on multiple servers with VMs. We focus on developing a flexible egalitarian scheme where all applications have the same priority, i.e., the target is to minimize their collective execution. While the problem of non-egalitarian allocation of resources to meet different application priorities is also very important, it is not the subject of this dissertation but part of our future work. The main contributions of this chapter are:

- A polynomial time algorithm is developed for the multi-dimensional VM resource placement problem, such that the maximum load over all resource dimensions and all servers is minimized. The algorithm removes the assump-

tion of cross-resource demand normalization required by previous work and outputs all valid VM consolidation strategies that achieve min-max load balancing on at least one resource.

- A consolidation performance prediction methodology that is based on a multi-class, closed queueing network, is proposed. This methodology moves beyond finding an assignment configuration where loads on the various physical components are equalized by encapsulating the effect of queueing (i.e., of overheads due to competition for resources) on performance, something that the mix-max algorithm for VM placement is oblivious of.

We evaluate the robustness of the proposed scheme using the RUBiS multi-tiered application in a virtualization environment that uses Xen and show that indeed the proposed solution is robust, i.e., it can automate the difficult process of multi-resource provisioning across multiple servers very effectively. We stress that the selection of RUBiS as a workload makes the problem more challenging as it requires good responsiveness for the three different tiers that are located on three different physical servers in order to minimize the application end-to-end response times.

6.1 Background: Fair Load Balancing on a Single Resource

We first present the fairness and load balancing in a general job scheduling problem where each of m users has to be assigned to a subset of n machines (i.e., each machine is considered as a *single* resource).

6.1.1 Max-Min Fairness

Given m users with resource demands $\langle D_1, D_2, \dots, D_m \rangle$, let $a_x = \langle A_x(1), A_x(2), \dots, A_x(m) \rangle$ be the allocated resource ranked in increasing order by scheme x . $A_x(i)$ is the i -th smallest component. For two allocation outputs a_x and a_y , a_x is said to have higher lexicographical value than a_y if there is an index j such that $A_x(j) > A_y(j)$ and $A_x(i) = A_y(i)$ for every index $i < j$. In all feasible allocations of a resource allocation problem, an allocation a^* is called max-min fair if it has the same or higher lexicographical value than any other feasible allocation.

6.1.2 Min-Max Load Balancing

Let n machines with load $\langle L_1, L_2, \dots, L_n \rangle$. Let $l_x = \langle L_x(1), L_x(2), \dots, L_x(m) \rangle$ be the allocated load by scheme x , ranked in decreasing order. $L_x(i)$ is the i -th largest component. For two allocation outputs l_x and l_y , l_x has lower lexicographical value than l_y if there is an index j such that $L_x(j) < L_y(j)$ and $L_x(i) = L_y(i)$ for every index $i < j$. In all feasible allocations of a load allocation problem, an allocation l^* is called min-max load balanced if it has the same or lower lexicographical value than any other allocation.

6.1.3 Fair Load Balancing

Fair load balancing ensures fair resource allocation among users through "balancing" of the machine load. With careful association between users and machines, fair load balancing targets max-min fairness in resource allocation among users and targets min-max load balancing on machine load. This can lead to provable system properties, e.g., Pareto efficiency [64], in data center resource management. Fair load balancing maximizes total user satisfaction when individual user satisfaction can be modeled by a concave function [116]. In VM consolidation where individ-

ual user satisfaction can be described by the virtualized application performance in a VM, fair load balancing can lead to optimal total system performance if the application performance is a concave function of allocated resources.

When a user can use resources from multiple machines simultaneously, it is called *multiple-association*. In this case, there is a strong correlation between min-max fairness and max-min load balancing. Bejerano et al. [38] proved the following property:

Lemma 1. *In the multiple-association case, a min-max load balanced assignment defines a max-min fair resource allocation and vice-versa.*

When a user can use resources from no more than one machine at any time (called *single-association*), Bejerano et al. [38] showed that a min-max load balanced assignment may still define a 2-approximation max-min fairness. Therefore, in the rest of the chapter, we aim at min-max load balancing, which can lead to optimal max-min fair resource allocation (approximation in *single-association*).

6.2 Fair Load Balancing on Multiple Resources: Challenges

Ghods et al. [64] propose the concept of dominant resource fairness, a generalization of max-min fairness to multiple resource scenario. For each user, the maximum among all resource shares allocated to that user is called his/her dominant share, and the resource corresponding to the dominant share is called the dominant resource. Dominant resource fairness allocation seeks for max-min fairness across users' dominant shares.

One problem with dominant resource fairness is on the multi-resource load balancing side. Since it considers only one resource for each user, there is no definition on how to handle the rest of the resources when load unbalancing on those

resources could still affect user performance. For example, in a simple VM consolidation scenario shown in Figure 6.1(a), there are four VMs (each is configured with 1 VCPU and 1GB-memory), each hosting one SPEC CPU2006 benchmark application [15], to be assigned onto two 2GB-memory 2-core servers. Two VMs host instances of *gobmk* and two host instances of *lbm*. The resource usage of the two SPEC applications is : *gobmk* (CPU: 100%, Mem: 15.8% (of 1GB)), *lbm* (CPU: 100%, Mem: 80.1% (of 1GB)).

All 4 VMs are CPU-intensive, and the two hosting *lbm* have also higher memory intensity than those hosting *gobmk*. In dominant resource fairness, CPU is the dominant resource for all VMs since each one has the maximal 100% share for it. However, consider two consolidation plans: P1, where the two *gobmk* VMs are consolidated onto one machine and the two *lbm* VMs onto another, and P2, where each machine hosts one *gobmk* VM and one *lbm* VM. Figure 6.1(b) shows that P2, a balanced placement on both CPU and memory usage, can significantly improve the performance: the job completion time of *lbm* is 34% less than in P1, while performance of *gobmk* is merely impacted. In the following section, we present a formalization of the problem and its proposed solution.

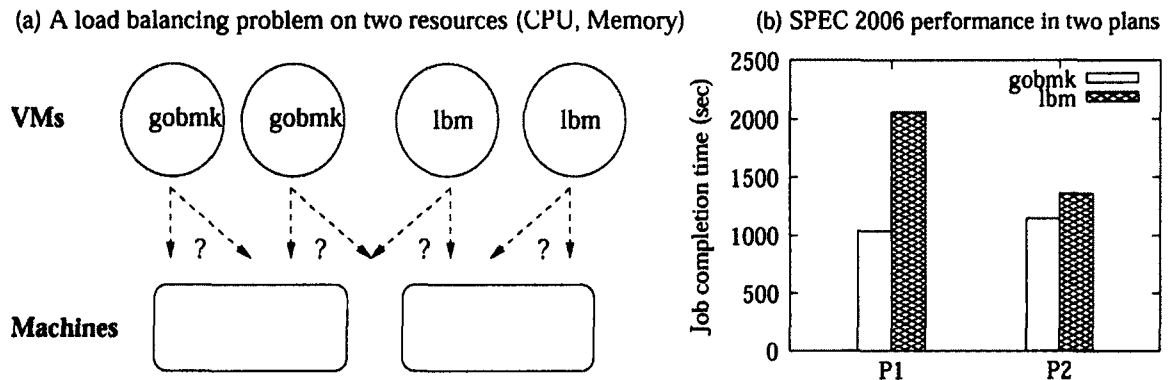


Figure 6.1: A simple load balancing problem on two resources (CPU, memory).

6.3 Fair VM Allocation Algorithm

We define a vector scheduling problem for fair load balancing in virtualized data centers. This problem has a set of n VMs with d -dimensional resource demand vectors $D_i = \langle D_i^1, D_i^2, \dots, D_i^d \rangle$, $1 \leq i \leq n$, and m homogeneous machines with resource capacity $C = \langle C^1, C^2, \dots, C^d \rangle$. The objective of the vector scheduling is to assign n VMs onto m machines and minimize $\max_{1 \leq i \leq m} \|\bar{A}_i\|_\infty$, where $\bar{A}_i = \sum_{j \in A_i} D_j$ is the sum of the VM demand vectors placed on machine i . That is, we seek min-max load balancing such that the maximum load over all dimensions and all machines is minimized. The multi-dimensional vector scheduling problem is NP-complete [48].

Our algorithm is an extension of the vector scheduling algorithm in [48]. The basic idea is a primary-dual approach where the scheduling problem is solved indirectly through a bin packing problem. By guessing the optimal solution for the vector scheduling problem as h , the scheduling algorithm aims to place all VMs (balls) into the machines (bins) of resource capacity h . The bin packing either is infeasible or can be done successfully. The scheduling algorithm takes this bin packing as a decision procedure to do a binary search for the optimal value h^* and the companion VM consolidation (bin packing) decision.

While [48] defined h as a scalar value, we define h as a vector $\langle h^1, h^2, \dots, h^d \rangle$, where d is the number of resources. We do not assume uniform normalization across different resources, and address the multi-dimensionality by integrating a performance prediction model with a polynomial time approximation scheme (PTAS).

6.3.1 Vector Bin Packing

The vector bin packing algorithm takes an error-tolerance parameter $\epsilon > 0$ and, in polynomial time, produces a solution that is within a factor $1 + \epsilon$ of being optimal.

Before presenting the algorithm, we first define a few terms:

- A *capacity configuration* is a d -tuple of integers $A = (a_1, a_2, \dots, a_d)$ such that $0 \leq a_i \leq \lceil \frac{h_i}{\epsilon} \rceil$. There are at most $t = \prod_{1 \leq i \leq d} (1 + \lceil \frac{h_i}{\epsilon} \rceil)$ capacity configurations.
- Given a capacity configuration A , its corresponding *empty capacity configuration* is the d -tuple obtained by subtracting each a_i from $(1 + \lceil \frac{h_i}{\epsilon} \rceil)$.
- A *bin configuration* is a t -tuple of integers $M = (m_1, m_2, \dots, m_t)$ such that $0 \leq m_i \leq m$ and $\sum_i m_i = m$.
- Given a bin configuration M , its corresponding \overline{M} is the one obtained by taking the empty capacity configurations for each i in M .
- A packing of VMs in a machine is said to fit a capacity configuration (a_1, a_2, \dots, a_d) if the sum of VM load in each resource dimension i is less than $\epsilon \cdot a_i$.

A capacity configuration describes approximately how a machine is filled in terms of aggregated load size and a bin configuration describes the number of machines of each capacity configuration. They together describe a load volume allocation scheme on the m machines. The algorithm consists of the following steps:

1. Reduce to zero all coordinates of the load vectors that are too small. Specifically, let $\delta = \frac{\epsilon}{d}$; for each VM i , $D_i^j = 0$ if $D_i^j \leq \delta \|D_i\|_\infty$. This step bounds the ratio of the largest coordinate to the smallest non-zero coordinate in VM load vectors, and helps the next step to discretize the large VMs into a small number of distinct-sized classes.
2. Classify each VM into either *small* or *large* based on their load vectors. Specifically, for each VM i , assign it into the *large* group if $\|D_i\|_\infty > \delta$; otherwise, it is assigned into the *small* group.
3. Pack all *large* VMs onto the machines as follows:

- (a) For each resource dimension i , notice the smallest non-zero coordinate of a *large* VM is at least δ^2 ; Discretize the load interval $[\delta^2, h^i]$ into $q = \lceil \log(1 + \epsilon)^{\frac{h^i}{\delta^2}} \rceil$ intervals of the form $(x_0, (1 + \epsilon)x_0]$, $(x_1, (1 + \epsilon)x_1]$, \dots , $(x_{q-1}, h^i]$, where $x_0 = \delta^2$ and $x_{i+1} = (1 + \epsilon)x_i$. Next, round each non-zero coordinate of a large VM's load vector down to the left end point of the interval in which it falls, and have the discretized load vectors for all large VMs .
- (b) For each defined bin configuration M , use a simple dynamic programming algorithm to decide if there is a packing of large VMs that fits M as follows: Order all m machines in some arbitrary way, and assigns a capacity configuration from M to each machine; for $1 \leq j \leq m$, compute recursively computes all possible subsets of the discretized load vectors from the large VMs that can fit into the first i machines; the dynamic programming algorithm ends at $j = m$, and either reports "no" on the proposed bin configuration M , or reports one valid solution on placing the large VMs according to M .

This step takes the fact of a logarithmic number of distinct-sized classes in large VMs and places them on the machines through dynamic programming.

4. Pack all *small* VMs in the small group on top of large VMs. Given a valid bin configuration M and each of its valid solution placing the large VMs onto the m machines, find an approximate feasible solution to pack the small VMs onto the valid solution in M as follows:

- (a) Assume the small group size is K , and number the VMs in the small group from 1 to K ;
- (b) Formulate the small VM packing as a linear programming problem with

the min-max load balancing optimization objective:

$$\begin{aligned} \sum_{j:1 \leq j \leq m} y_{ij} &= 1, 1 \leq i \leq K \\ \sum_{i:1 \leq i \leq K} D_i^d y_{ij} &\leq b_j^d, 1 \leq j \leq m, 1 \leq d \leq D \\ y_{ij} &\geq 0 \end{aligned}$$

where b_j^d is the height bound for each machine j and the resource dimension d defined in the bin configuration \overline{M} .

- (c) After solving the above linear program, let S' be the set of small VMs that are assigned to more than one machine. Partition the set S' into m subsets of at most d VMs each in a uniform random way and assign each subset to each machine accordingly.

This step places the small VMs on top of the large VMs using a linear programming relaxation and a careful rounding to avoid multiple associations.

6.3.2 Discussion

Step 3 dominates the time in the whole process. There are approximation algorithms that can speed up this step, such as the linear programming relaxation and rounding approach used in Step 4, but the approximation factor is usually large [38].

Practical constraints in VM placement problems such as affinity rules, VM-based machine subset specification, can be introduced in the dynamic programming of Step 3 and the linear programming of Step 4.

While we assume all VMs have the same priority, there are consolidation scenarios where some VMs are considered more important than others when there is a resource contention. Then, a possible approach is to assign weights to VMs based

on their priorities. A VM's weight can be multiplied by its resource demand so that higher-priority VMs have inflated resource demand. While this inflation might lead to non-feasible solution in server physical resource capacities, min-max load balancing is not affected when server virtual capacities are introduced, as the guessed optimal solution h itself is also a server virtual capacity.

6.3.3 Vector Scheduling with Predictive Model

Our vector scheduling algorithm consists of three steps:

1. Given an $\epsilon > 0$ and a guess for the optimal maximum load vector h , call the vector bin packing algorithm with ϵ and h , which either returns a VM consolidation of maximum load $(1 + \epsilon)h$, or proves that the guess h is infeasible.
2. Repeat step 1 through a multi-dimensional binary search for the optimal value h^* . The multi-dimensional binary search is done as follows:
 - (a) for each dimension i , the maximum load vector guess starts with $\langle C^1, \dots, h^i, \dots, C^d \rangle$;
 - (b) use binary search to find the minimal h^{i*} which can lead to at least one valid solution with the vector bin packing algorithm and the maximum load constraint vector $\langle C^1, \dots, h^{i*}, \dots, C^d \rangle$;
 - (c) use binary search to find the minimal α^{i*} ($0 < \alpha^{i*} \leq 1$) which can lead to at least one valid solution with the vector bin packing algorithm and the maximum load constraint vector $\langle \alpha^{i*}C^1, \dots, h^{i*}, \dots, \alpha^{i*}C^d \rangle$;
 - (d) record (h^{i*}, α^{i*}) and the companion solutions as successful VM consolidations for dimension i .
3. Predict the consolidation performance of the successful solutions from Step 2 and output the VM consolidation scheme with the best performance.

For the guess of optimal maximum load vector h^i on each dimension i , there is a naive lower bound of 0. We can use tighter low bounds such as the maximal value of the VM load vectors along dimension i , or the average VM load on that dimension.

Theorem 2. *Given any fixed $\epsilon > 0$, our algorithm is a $(1 + \epsilon)$ approximation algorithm for the vector scheduling problem that runs in $O(dm \ln \frac{1}{\epsilon} (\frac{nd}{\epsilon})^{O(s)})$ time, where $s = O((\frac{\ln(d/\epsilon)}{\epsilon})^d)$.*

Proof. Step 1 of the vector bin packing algorithm will increase the load of the machines by only a $(1 + \epsilon)$ factor after restoring those reduced-to-zero load coordinates back to their original values; Step 2 of the vector bin packing algorithm also brings only a $(1 + \epsilon)$ factor since each discretized coordinate value is at least $(1 + \epsilon)^{-1}$ times the original values; in Step 3 of the vector bin packing algorithm, each small VM has every coordinate value $\leq \frac{\epsilon}{d}$ and any machine is assigned at most d small VMs, therefore Step 3 does not violate the machine load by more than ϵ in any dimension. Overall, the vector bin packing algorithm yields a result with $(1 + \epsilon)$ -approximation.

On the polynomial time complexity, we assume that the resource capacity C and dimensions d are constants. The run time is dominated by Step 2 for packing large VMs. There are at most $m^t = O(n^{O(\epsilon^{-d})})$ bin configurations; for each bin configuration, the running time of the dynamic programming algorithm is $O(m(\frac{nd^2}{\epsilon})^s)$, where $s = (1 + \lceil \frac{2}{\epsilon} \ln \delta^{-1} \rceil)^d$ is the distinct discretized load vectors in Step 2. Therefore, the time for completing one run of the vector bin packing algorithm is $(\frac{nd}{\epsilon})^{O(s)}$, while the multi-dimensional binary search takes $O(d \ln \frac{1}{\epsilon})$ guesses to finish. \square

Next, we present a queueing-network based methodology for application performance prediction that is required in Step 3 of the vector scheduling algorithm. The queueing network prediction is necessary to incorporate the queueing effect (due to

co-location and competition for common resources) of the multiple VMs. This effect is not captured by the min-max allocation which is the core of the fair allocation algorithm and it is necessary to select from the multiple allocation options the most fitting one.

6.4 A Predictive Queueing Model

Closed queueing network models have been successfully used in modeling multi-tier applications on shared physical resources (e.g., network, disk) [98, 130]. In this section, we propose a consolidation performance prediction methodology based on a closed queueing network model.

The vector bin packing algorithm significantly reduces the number of candidate schemes for the queueing model prediction step. Theoretically, the number of ways to place n distinct VMs into m identical hosts with r_0 empty hosts, r_1 hosts containing 1 VM, r_2 hosts containing 2 VMs, r_n hosts containing n VMs [108] is: $\frac{n!}{(1!)^{r_1} r_1! (2!)^{r_2} r_2! \dots (n!)^{r_n} r_n!}$. For example, if one is to place $n = 6$ VMs among $m = 3$ hosts and $r_2 = 3$ hosts contain 2 VMs (all other r_i s are zero), the number of placement is $\frac{6!}{(2!)^3 3!} = 15$. If $n = 9$ VMs is placed into $m = 3$ hosts and $r_3 = 3$ hosts contain 3 VMs, the number is 280.

6.4.1 RUBiS Multi-tiered Benchmark

RUBiS [45] is an auction site prototype modeled after eBay.com and is widely used in performance studies of system and multi-tiered applications. In the RUBiS implementation, there are three tiers of servers: the Apache Web server, the EJB server, and the MySQL relational database. They usually reside in three different virtual machines.

In Figure 6.2, we show the closed queueing network model to represent the

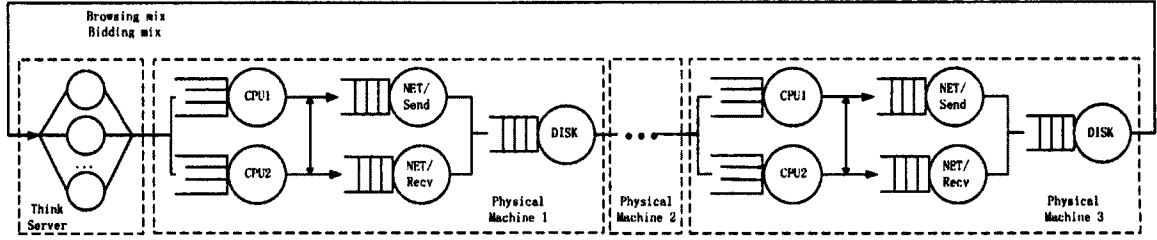


Figure 6.2: Closed queueing network model for multi-tier server and multi-class workload

RUBiS three-tier application. Three dashed boxes are used to abstract the web server, the application server, and the database server, respectively. The think server represents the average client think time Z between receiving a web page and submitting a new page request.

6.4.2 Application Profiling: Service Demand Estimation

To utilize the queueing model, it is critical to accurately measure the application average total service demands per resource, i.e., the total time an application spends on each resource excluding the time spent waiting to gain access to a resource due to contention. This metric is called *average service demand* [97]. Since the demand is the total service time at a device, it can be thought of as a product of the average number of "visits" to the device and the average amount of time required at the device per visit. We define the following average service demands for applications as follows:

- S_c : average CPU service demand, i.e., average CPU time during one execution of a workload class.
- S_n : average network service time, i.e., average networking time during one execution of a workload class. For modeling accuracy, we subdivide this metric into two metrics --- the average sending service time $S_{n,s}$ and the average receiving service time $S_{n,r}$.

- S_d : average disk service demand, i.e., aggregate disk I/O service time during one execution of a workload class.

There are related studies focusing on obtaining resource demands of a single application using fine-grained and detailed information [49]. To this end, the standard approach is to explore a large number of hardware counters or even modify the application source code, a process that is cumbersome, intrusive, and neither portable nor scalable. Using hardware performance counters is clearly the preferable approach but it is challenging to collapse such diverse information into a single value parameter like the resource demand. In the following four sections, we illustrate how we achieve the above targets via a light-weight profiling methodology that can capture the average resource demands. A similar approach has been used to model Java workloads [31, 50].

6.4.2.1 Assumptions

We now describe two assumptions of our proposed queuing network model. For simplicity, we assume that each VM hosting an application is assigned to one core on a multi-core server consistent with the literature where each VM runs on a separate core [67, 77], but the proposed approach is not restrained by this assumption [50]. Second, we assume that the performance degradation due to memory and cache contention during consolidation is captured via the average CPU service demand S_c . Prior works have shown that it is not necessary to explicitly model memory and cache, this can be implicitly modeled via the distribution of the service process (i.e., the service demand) [98] or using load dependent service times [50, 97].

6.4.2.2 Per-tier Service Time

To measure the per-tier service time, we utilize packet level capturing and filtering tools to extract the service time of each tier of RUBiS servers. The response time

of each tier is the time from the moment the last packet of the request arrives to the moment the last packet of the response is sent. `tcprstat` [17] is an open source tool to capture and analyze packets to extract the delay between requests and responses, during a measurement interval. The tool can be set to monitor network traffic on a specified port, which makes it suitable for timing requests and responses to a single daemon process such as `mysql`, `httpd`, or any of other server processes. We modified the original code to store the response-time statistics in a central database for every measurement interval. This change allows `PREMATCH` to query the response information in real time. Figure 6.3 shows the an example of query result of response-time information collected.

| timestamp | hostname | count | max | min | avg | med | stddev | max_95 | avg_95 | std_95 | max_99 | avg_99 | std_99 |
|---------------------|----------|-------|---------|------|--------|------|---------|--------|--------|--------|--------|--------|--------|
| 2013-07-29 15:53:31 | web | 97 | 95806 | 1195 | 4176 | 1281 | 11802 | 26748 | 1935 | 3765 | 39981 | 3221 | 7238 |
| 2013-07-29 15:53:30 | app | 102 | 95005 | 506 | 4003 | 669 | 12622 | 25973 | 1340 | 3690 | 39149 | 2566 | 7062 |
| 2013-07-29 15:53:37 | db | 29 | 39169 | 98 | 7734 | 366 | 10550 | 20130 | 5693 | 7619 | 31397 | 6611 | 8873 |
| 2013-07-29 15:53:51 | web | 122 | 57543 | 1155 | 3693 | 1256 | 9054 | 25879 | 1642 | 2674 | 41399 | 2917 | 6762 |
| 2013-07-29 15:53:50 | app | 189 | 42194 | 496 | 2596 | 634 | 7456 | 25060 | 982 | 2731 | 36659 | 2181 | 6314 |
| 2013-07-29 15:53:57 | db | 35 | 8880265 | 96 | 261222 | 356 | 1478183 | 21040 | 6983 | 8607 | 32088 | 7721 | 9482 |

Figure 6.3: An example of `tcprstat` query result

6.4.2.3 Average Network Demand

Measuring the demands of the network is done by monitoring the collected network traffic issued from guest VMs in two metrics: `rxbyt/s` (bytes received per second) and `txbyt/s` (bytes transmitted per second). These two metrics can be obtained by running the `sar` utility [18] in the guest domain or the `proc` filesystem at `/proc/net/dev` in Xen driver domain.

To obtain the average network service demand, the following measurements are required to collect during the execution of the application:

- t -- total execution time.
- λ_s, λ_r -- average sending/receiving network traffic rate of the guest VM.

- n -- the number of requests issued by the clients during time t .
- R -- transmission rate of the Ethernet link.

The average service demand for network is computed as follows:

$$S_{n,s} = \frac{\lambda_s t}{nR}, \quad S_{n,r} = \frac{\lambda_r t}{nR} \quad (6.1)$$

6.4.2.4 Average Disk Demand

Measuring the performance of the disk is particularly difficult due to the multiple buffers present at all storage levels, the high number of run time optimizations, e.g., out-of-order writes, and parallel writing across different disk platters. Profiling of disk demand must incorporate the inherent disk parallelism due to consolidation.

Here, we rely on disk operation statistics, instead of disk utilization, for the following reasons. First, to measure/compute disk utilization, one has to know its capacity. However, for disk operations, this value differs under workloads, e.g., for sequential read/write and random read/write. Second, the utilization is a biased metric due to possible I/O buffering. To compute the disk demand S_d , we need to figure out the disk parallelism and total disk execution time. We apply a method similar to the one in [31] that uses the average service queue size, denoted by q , as an indicator of the disk parallelism. For the total disk execution time, we use the product of the average disk operations per second (i.e., the disk throughput), denoted by λ_d , and the average service time, denoted by s , which is obtained by the `iostat` utility [7]. S_d is estimated as the total disk time divided by the disk parallelism:

$$S_d = \frac{\lambda_d s}{q + 1} \quad (6.2)$$

6.4.2.5 Average CPU Demand

To compute the CPU demand per tier, we follow the steps:

1. Measure the per-tier service time T via the `tshark` utility.
2. Compute the average network demand $S_{n,s}$ and $S_{n,r}$ using Eq. (6.1).
3. Compute the average disk demand S_d using Eq. (6.2).
4. Deduce the average CPU demand as

$$S_c = T - S_{n,s} - S_{n,r} - S_d. \quad (6.3)$$

By this method, S_c implicitly includes the service demands of memory and cache. This simplification is accurate for our purposes.

6.5 Evaluation

We evaluate the effectiveness and accuracy of our algorithm and model with the RUBiS application. The testbed runs the Fedora release 8 operating system with Linux kernel 2.6.18-8. The evaluation is based on the Xen [34] virtualization platform version 3.3.1. Our testbed platform uses Supermicro 1U Superservers with Intel Core 2Duo E4300 1.86GHz, 2MB L2 cache. All servers have a RAM of 2GB and 250GB 5400RPM disk. The servers are connected through D-LINK DES-3226L 10/100Mbps switches. PREMATCH is implemented on Usher, a virtual machine management framework developed by McNett et al [96]. Figure 6.4 shows the overview of the PREMATCH architecture. It makes the fair load balancing consolidation decision through the two-step mechanism and handles it to Usher for the VM placement execution.

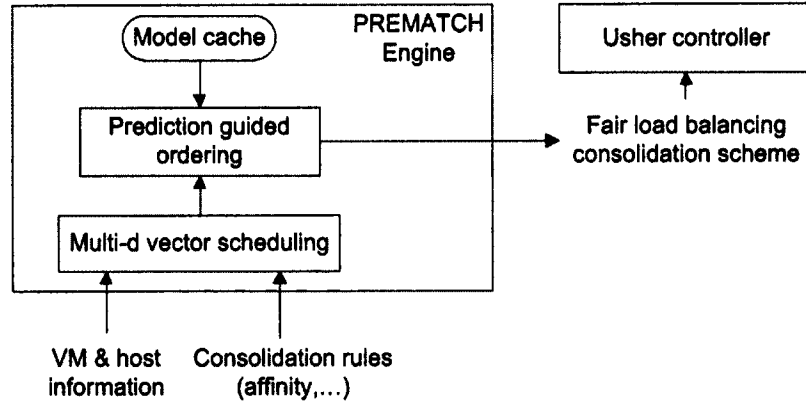


Figure 6.4: PREMATCH architecture graph.

Table 6.1: Profiling of browsing and bidding workload

| Demand | Browsing | | | Bidding | | |
|-----------|----------|-------|--------|---------|-------|---------|
| | web | app | db | web | app | db |
| S_c | 5.2ms | 4.8ms | 10.3ms | 3.7ms | 4.8ms | 25.5 ms |
| $S_{n,s}$ | 5.8ms | 5.2ms | 0.2ms | 4.6ms | 4.4ms | 0.2ms |
| $S_{n,r}$ | 5.7ms | 0.6ms | 0 | 4.7ms | 0.6ms | 0 |
| S_d | 0 | 0 | 0 | 0 | 0 | 2.2ms |

Typically, RUBiS has two different workload mixes: the browsing mix and the bidding mix. The browsing mix includes browsing items, consulting the bid history and obtaining seller information. The browsing mix is made up of only read-only interactions. The bidding mix includes 15% write interactions and is considered the most representative of an auction site workload according to an early study of the eBay [44]. We summarize the estimated average CPU, network, and disk demands in Table 6.1. Note that the bidding mix has an average of 2.2 ms demand on disk writing operation because of saving bidding, buying, or selling items, leaving comments on other users. Disk usage for the browsing mix is initially observable in order to load records into memory, but then drops off to a very low value and is therefore neglected. One can see that the two workloads have different properties: for the bidding mix the database, CPU is the bottleneck (25.5 ms), while for the browsing mix the network is the bottleneck ($5.7 + 5.8 = 11.5$ ms).

6.5.1 RUBiS with Different Number of Clients

To demonstrate the accuracy of the profiling technique in Section 6.4.2, Figure 6.5 reports both measured and modeled average response times and throughput for RUBiS as a function of the number of clients for the two workloads. The number of emulator client changes from 200 to 1200 for the browsing mix and from 100 to 400 for the bidding mix. Each run takes 30 minutes and we average three runs of RUBiS for each data point. The think times of the browsing and the bidding mix are set to 7 and 4.5 seconds respectively, which are the default values. Both figures illustrate that the profiling technique and queuing network model have a good accuracy in predicting the RUBiS performance for different client numbers. The difference of the two workloads are shown in Figure 6.5(b) where the bidding mix reaches the throughput bound at 41 interactions per second while the browsing mix reaches the maximum throughput at 102.

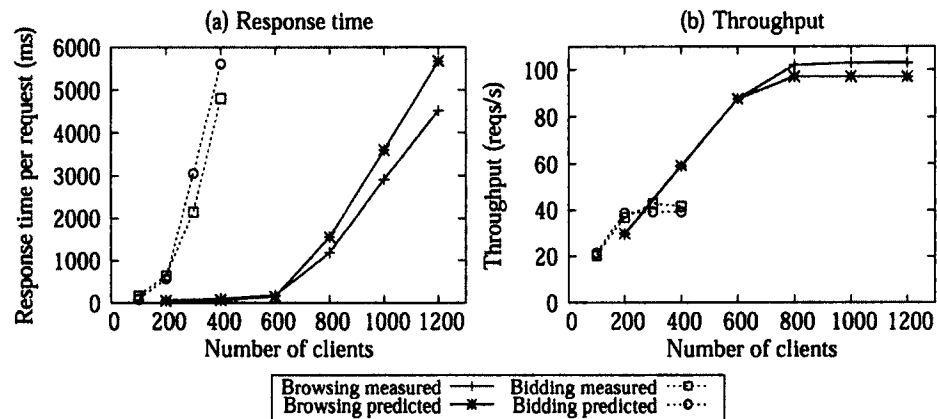


Figure 6.5: Measured vs. predicted average response time and throughput.

6.5.2 Browsing and Bidding Mix Consolidation

With the assumptions in Section 6.4.2, consolidating two RUBiS applications (one browsing and one bidding workload) on three machines leads to 15 consolidation schemes (see Table 6.2). To understand the "settings" column, consider W_1A_2--

$A_1D_2--D_1W_2$ as an example. The subscripts denote which RUBiS application the virtualized server belongs to, e.g., W_1 means virtualized webserver of RUBiS application 1 (browsing mix). The dashes "--" separate each physical server and group the VMs on the same machine, e.g., " W_1A_2-- " means a configuration that has the virtualized webserver of RUBiS application 1 and the virtualized appserver of RUBiS application 2 on the same machine. It has three groups of consolidated VMs since three physical machines are used in the experiments.

Table 6.2: Configuration table

| Cfg# | Settings | Cfg# | Settings | Cfg# | Settings |
|-------|--------------------------|--------|--------------------------|--------|--------------------------|
| Cfg 1 | $W_1W_2--A_1A_2--D_1D_2$ | Cfg 6 | $W_1D_2--A_1A_2--D_1W_2$ | Cfg 11 | $W_1D_1--A_1A_2--W_2D_2$ |
| Cfg 2 | $W_1A_2--A_1W_2--D_1D_2$ | Cfg 7 | $W_1A_1--D_1D_2--W_2A_2$ | Cfg 12 | $A_1D_1--W_1W_2--A_2D_2$ |
| Cfg 3 | $W_1A_2--A_1D_2--D_1W_2$ | Cfg 8 | $W_1A_1--D_1A_2--W_2D_2$ | Cfg 13 | $W_1D_1--A_1D_2--W_2A_2$ |
| Cfg 4 | $W_1W_2--A_1D_2--D_1A_2$ | Cfg 9 | $W_1A_1--D_1W_2--A_2D_2$ | Cfg 14 | $W_2A_2--W_1D_2--A_1D_1$ |
| Cfg 5 | $W_1D_2--A_1W_2--D_1A_2$ | Cfg 10 | $W_1D_1--A_1W_2--A_2D_2$ | Cfg 15 | $W_1D_1--A_1W_2--A_2D_2$ |

For the first scenario, we run two classes of workloads on client emulators: the first one uses the browsing mix with 600 clients and a thinking time of 7 seconds (default value) and is denoted as **application 1**. The second uses the bidding mix with 200 clients and 4.5 seconds (default value) thinking time and is denoted as **application 2**. PREMATCH takes the following steps:

Step 1: Measure multi-dimensional resource demands. We collect the resource baseline usage for each tier of RUBiS when it is virtualized and runs in one physical machine *alone*. It serves as the input to the bin packing algorithm described in section 6.3.1. We show the multi-dimensional demands of workloads in Table 6.3.

Step 2: Run the vector bin packing algorithm for each dimension. The vector bin packing algorithm takes two sets of vector loads L_{web} , L_{app} , and L_{db} (in total of 6 demands vectors) as inputs. It outputs the min-max load balancing configuration and association for each resource as shown in Table 6.4. Take the first row in Table 6.4 as an example: it shows that when considering the CPU resource, the al-

Table 6.3: Service loads for three workloads

| | Bidding mix 200 | | | Browsing mix 600 | | | Browsing mix 800 | | |
|-----------------|-----------------|-----------|----------|------------------|-----------|----------|------------------|-----------|----------|
| | L_{web} | L_{app} | L_{db} | L_{web} | L_{app} | L_{db} | L_{web} | L_{app} | L_{db} |
| CPU | 6.2% | 7.3% | 44.7% | 18.5% | 17.0% | 44.3% | 21.6% | 19.9% | 55.1% |
| Net Send | 13.7% | 1.6% | 0.1% | 39.9% | 37.2% | 1.4% | 45.6% | 42.6% | 1.6% |
| Net Recv | 13.8% | 13.0% | 0.7% | 40.8% | 4.35% | 0.2% | 46.5% | 5% | 0.3% |
| Disk | 0.03% | 0.06% | 0.3% | 0.39% | 0.06% | 0.04% | 0.44% | 0.07% | 0.04% |
| Memory | 5.8% | 23.6% | 31.9% | 36.9% | 23.6% | 15.9% | 31.4% | 23.5% | 15.5% |

Table 6.4: Min-max load balancing for the bidding and browsing mix consolidation

| Min-max config | Machine1 | | | |
|------------------|--------------|--------------|--------------|--------------|
| | CPU | Net | Disk | Mem |
| Cfg3 on CPU | 25.9% | 47.6% | 0.45% | 60.6% |
| Cfg14 on NET | 14.2% | 21.1% | 0.09% | 29.4% |
| Cfg1 on DISK/MEM | 25.3% | 54.2% | 0.42% | 42.7% |
| | Machine2 | | | |
| | CPU | Net | Disk | Mem |
| | 60.9% | 21.1% | 0.36% | 55.6% |
| | 62.4% | 40.7% | 0.36% | 69.8% |
| | 24.4% | 28.0% | 0.12% | 47.3% |
| | Machine3 | | | |
| | CPU | Net | Disk | Mem |
| | 51.1% | 14.6% | 0.07% | 21.7% |
| | 61.3% | 21.6% | 0.43% | 39.6% |
| | 88.2% | 1.1% | 0.43% | 47.9% |

gorithm outputs configuration 3 (see Table 6.2). The following readings in the same row denote the estimated resource consumption. We highlight the computed min-max value of the corresponding resource (the boldfaced, underlined value). The results show that three different configurations achieve min-max load balancing. In the next step, we use the predictive queuing model to compare the performance of each configuration and choose the best one as final decision.

Step 3: Queuing model prediction. We plug both bidding and browsing profiling parameters from Table 6.1 into the queuing model shown in Figure 6.2. We apply mean value analysis (MVA) [97] to solve the model and get the predicted response time and system throughput for each application in Table 6.5.

Table 6.5: Predicted performance for min-max configs

| Min-max config | App1 (Browsing) | | App2 (Bidding) | |
|------------------|----------------------|----------------------|----------------------|----------------------|
| | Response Time | Throughput | Response Time | Throughput |
| | Predicted (Measured) | Predicted (Measured) | Predicted (Measured) | Predicted (Measured) |
| Cfg3 on CPU | 162(169) ms | 83.7(83.2) req/s | 689(704) ms | 38.5(38.2) req/s |
| Cfg14 on NET | 122(150) ms | 84.2(83.9) req/s | 775(745) ms | 37.9(38.2) req/s |
| Cfg1 on DISK/MEM | 184(190) ms | 83.5(83.1) req/s | 896(1099) ms | 37.0(35.5) req/s |

Figure 6.6 illustrates that the predicted configuration indeed gives the best performance among all possible configurations in real measurements. Table 6.2 shows the details of each configuration. Figure 6.6 also shows the comparison of measured and predicted values for *all* configurations. With few exceptions, the model prediction is in excellent agreement with experimental data.

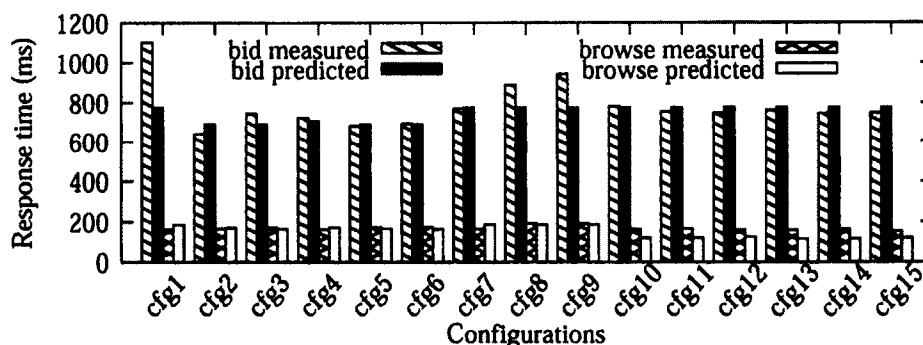


Figure 6.6: Measured vs. predicted average response time for browsing and bidding mix.

For each resource, Figure 6.7(a) compares the maximum load of the three machines for the PREMATCH configuration and the worst measured configuration. One can see that, for the network resource, the worst configuration consumes twice as much as the PREMATCH. The maximum memory usage of worst configuration is 13% more than PREMATCH's.

Figure 6.7(b) illustrates a side-by-side comparison of the measured performance of the PREMATCH, random, and worst configuration schemes. By random configuration, we run a naive randomized algorithm 100 times, which randomly picks one of the 15 configurations. Hence, the data we show is averaged across all outputs. The PREMATCH configuration reduces the worst configuration response time by

46% for bidding and 17.9% for browsing. It also reduces the random configuration response time by 24.5% for bidding and 11.7% for browsing.

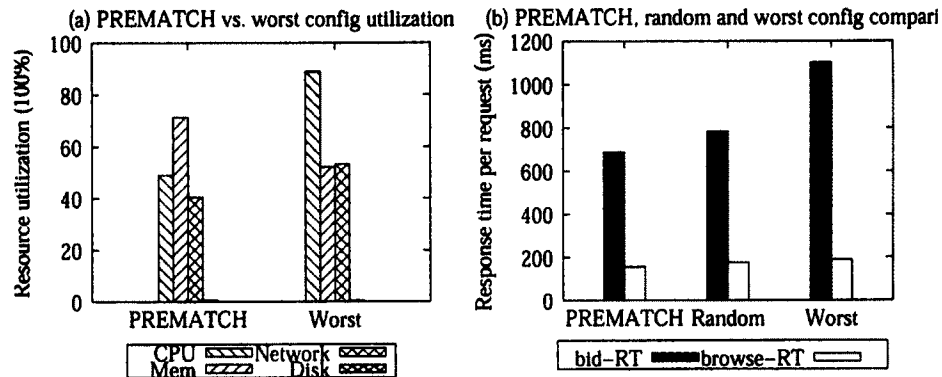


Figure 6.7: PREMATCH, random, and worst consolidation comparison.

6.5.3 Browsing Mix Consolidation

In this scenario, we give another example of consolidation of two browsing mixes. We have the same testbed setup as the one described in Section 6.5.2, but we run two client emulators with the same workload: two instances of the browsing mix with 800 clients and 7 seconds thinking time.

Table 6.6 contains the vector bin packing algorithm's configurations for each resource. Notice that, for this scenario, we have two candidate configurations achieving min-max load balancing for each resource because of the two identical workloads (two browsing mix with 800 clients). Calculated min-max value over all resources are highlighted (boldfaced and underlined).

The MVA results of estimated response time and system throughput for each application are in Table 6.7. The model selects "Cfg 6" and "Cfg 11" as the ones with best performance. In Figure 6.8, we present a side by side comparison of measured and predicted response time of 15 possible configurations. "app1" denotes the first RUBiS application and "app2" the second. The figure shows the accuracy of the model and corroborates the fact that the queueing model always selects the

Table 6.6: Min-max load balancing for browsing mixes consolidation

| Min-max config | Machine1 | | | |
|-----------------------|------------|--------------|--------------|------------|
| | CPU | Net | Disk | Mem |
| Cfg4 on CPU | 43.2% | 92.1% | 0.88% | 62.7% |
| Cfg12 on CPU | 75% | 24.75% | 0.88% | 39% |
| Cfg6 on NET/DISK/MEM | 76.7% | 47% | 0.48% | 46.9% |
| Cfg11 on NET/DISK/MEM | 76.7% | 47% | 0.48% | 46.9% |
| | Machine2 | | | |
| | CPU | Net | Disk | Mem |
| | 75% | 24.75% | 0.11% | 39% |
| | 43.2% | 92.1% | 0.11% | 62.7% |
| | 39.8% | 47.6% | 0.14% | 47% |
| | 39.8% | 47.6% | 0.14% | 47% |
| | Machine3 | | | |
| | CPU | Net | Disk | Mem |
| | 75% | 24.75% | 0.11% | 39% |
| | 75% | 24.75% | 0.11% | 39% |
| | 76.7% | 47% | 0.48% | 46.9% |
| | 76.7% | 47% | 0.48% | 46.9% |

Table 6.7: Predicted performance for min-max configs

| Min-max config | App1 | | App2 | |
|-----------------------|---------------|------------|---------------|------------|
| | Response Time | Throughput | Response Time | Throughput |
| Cfg4 on CPU | 2568 ms | 86.2 req/s | 2568 ms | 86.2 req/s |
| Cfg12 on CPU | 2568 ms | 86.2 req/s | 2568 ms | 86.2 req/s |
| Cfg6 on NET/DISK/MEM | 1616 ms | 96.1 req/s | 1616 | 96.1 req/s |
| Cfg11 on NET/DISK/MEM | 1616 ms | 96.1 req/s | 1616 | 96.1 req/s |

optimal prediction.

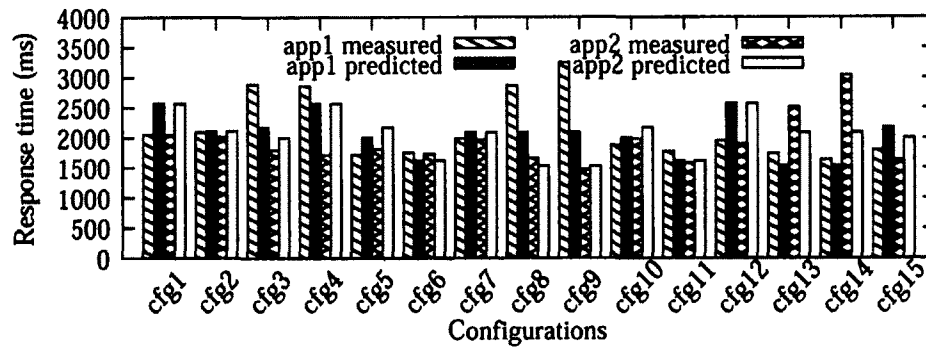


Figure 6.8: Measured vs. predicted average response time for the browsing mixes.

Figure 6.9(a) compares the maximum allocated load of three machines for our PREMATCH configuration and the worst measured configuration. For maximum network resource, the worst configuration consumes twice as much as PREMATCH. The maximum memory usage of worst configuration is 12% more than PREMATCH. Figure 6.9(b) illustrates a side-by-side comparison of the measured performance of PREMATCH, random, and worst configuration schemes. On average, the PREMATCH configuration reduces the worst configuration response time by 25.6% and reduces the random configuration response time by 13.7%. Meanwhile, for the worst configuration, the performance imbalance of the two applications results in poor performance.

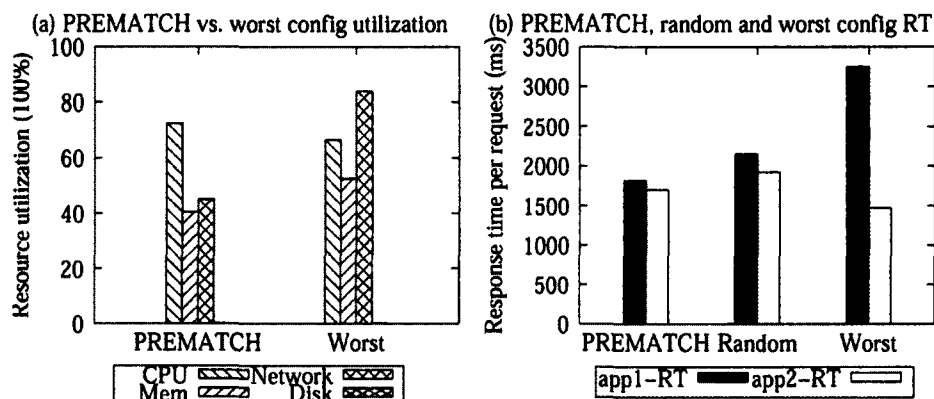


Figure 6.9: PREMATCH, random, and worst consolidation comparison.

6.6 Summary

In this chapter, we propose a method to assign VMs whose per-resource demands for multiple resource types (CPU, disk, etc.) are known to underlying hardware in a min-max fair way. Conflicts between min-max fairness of different resource types are resolved by using a queuing network model of each application to determine which resource should be prioritized. The experiment results with RUBiS applications demonstrate that the approach achieves good performance in consolidation decision making.

7 Summary of Contributions and Future Work

The main contribution of this dissertation is on the design of automatic system management with virtualized data centers. To this end, a set of new techniques and tools are developed and are summarized as follows.

- For application workload management, this dissertation describes and analyzes a session and request admission algorithm for session-based web services. The algorithm accepts requests until a threshold of active requests is reached. When this threshold is reached, incoming requests from newly starting sessions are dropped, while requests from already active sessions are queued. Only when this queue is empty and the number of active sessions is below the threshold, requests from new sessions are accepted for service again (in a more aggressive version, the system has to become idle before starting to accept new sessions again).

Simulation experiments with bursty session arrival patterns show that the above algorithm can reduce the number of dropped active sessions at the price of drops of newly starting sessions and increased response time. As the size of the request queue controls this trade-off, we propose an adaptive algorithm in which the request queue size is adapted in order to meet a certain response time requirement. We also show illustrations of the behavior of

the adaptive algorithms in different scenarios using the TPC-W e-commerce benchmarks.

- For server measurement, this dissertation presents a signal processing-based technique to estimate the amount of physical resources utilized by consolidate VMs. The dissertation first argues that profiling resource utilization from inside a VM does not always lead to accurate estimate. It then formulates the resource utilization profiling problem as a source separation problem studied in the area of digital signal processing. A directed factor graph model is defined to model the dependencies among multiple types of resources across physical and virtual layers. The results are presented from case studies that involve a diverse suite of applications, indicating improved accuracy of resource utilization estimates.
- For resource management, the dissertation presents AppRM, a system that automatically sets various resource control knobs provided by VMware vSphere in order to meet service level objectives. AppRM consists of various parts. The vApp manager collects statistics of the VMs comprising an application and builds a performance model of the application. The Application Controller in the vApp manager then figures out what resources the application needs in order to meet its SLO, and the Resource Manager translates these needs into actually vSphere settings. The Resource Pool Manager deals with the desires of multiple applications, making tradeoffs if needed or allocating more resources if available. The system is evaluated on two machines. One machine runs one or more MongoDB instances in a collection of VMs, while the other runs one or more workload generators in VMs. Each machine also runs AppRM in a VM. A large variety of experiments shows that AppRM is effective in achieving the SLOs even as loads change over time and compet-

ing VMs are using up resources as well. AppRm is implemented within the current release of VMware's vSphere. The current system deals only with CPU and memory resource as vSphere does not provide knobs for I/O and networking.

- For server consolidation, this dissertation addresses the multi-dimensional mapping problem for data centers, considering the CPU, disk, and network dimensions. It proposes a VM consolidation mechanism, first by extending the fair load balancing scheme for multi-dimensional vector scheduling and then by using a queueing network simulation to capture contentions for a particular VM placement. Experiments demonstrate that this approach indeed achieves good performance. The evaluation shows that it is useful to take into account VM consolidation when mapping VMs, since there can be large differences between best and worst configurations if interference is not taken into account.

7.1 Future Work

There are several extensions to the results presented in this dissertation.

7.1.1 Autonomic Resource Allocation

- **Scalability:** We do not have experimental results that demonstrate the scalability of AppRM due to the limited hardware we had access to. However, we deal with scalability in the design of AppRM using the following mechanisms: a) separation of concerns: AppRM utilizes a two-layered approach where vApp Managers translate per-vApp SLOs to desired VM-level resource settings and RP Managers determine actual resource settings based on available capacity; b) decentralized design: AppRM does not use a centralized

controller that determines resource allocations for everyone, which would not scale. Instead, each vApp Manager deals with only one vApp and each RP Manager handles only one resource pool; c) asynchronous communication: The RP Manager interacts with multiple vApp Managers asynchronously and determines value settings using Algorithm 2 thus markedly improve the scalability compared with previous work. On vSphere systems today, each DRS cluster can have up to 3000 VMs on 32 hosts and multiple RPs. Nevertheless, the number of settings each vApp Manager controls is only proportional to the number of VMs in that vApp and is not affected by the overall inventory size. In the extreme case, each RP Manager can take inputs from thousands of vApp Managers. However, the current DRS algorithm already performs fairly sophisticated computation for up to thousands of VMs every five minutes, hence, AppRM should not pose a scalability bottleneck given that it also runs on the order of minutes and does only simple calculation with the inputs. We plan to validate the scalability of AppRM empirically in future work.

- **Handling storage and network resources:** Network and storage control knobs are not fully implemented in VMware vSphere product at the time of this project. However, the modeling techniques used in our work can be applied to other resources as well. For storage, research prototypes [70,71] can provide I/O operations per second (IOPS) reservations and resource pools. For network, though bandwidth reservations for outgoing bandwidth are available, we do not know of any implementation of network resource pools. As these mechanisms mature, we hope to extend our work to other resources.
- **AppRM across virtual data centers:** Virtual data centers provide good abstraction of aggregation of resource pools, but AppRM cannot perform the resource control tuning, if not enough resources are allocated to a virtual data

center. AppRM can be extended to make recommendations to administrator and can also be extended to flow resources from a lower priority virtual data center, similar to the approach described in Chapter 5 for resource pools.

7.1.2 VM Auto-scaling

In Chapter 5, we demonstrate that dynamically adjusting the effective "size" of individual servers can successfully meet application SLO target. Another approach to solve the same problem is to change the number of servers (physical or virtual) hosting an Internet service or a multi-tiered application. A natural extension is to develop a system mechanism to add or remove an VM instance for one tier of a multi-tiered system while maintaining the system functionality. A new model is needed to determine the number of VM instances of a specific tier in order to meet a target SLO for a multi-tiered application. We will extend the current feedback control model or explore other methods, e.g., a queueing model or reinforcement learning model. By online monitoring the arrival flow and its statistics, the model should appropriately compute the number of virtual machines required at each tier. For instance, if a burst of arrivals is observed, then the system may need to add more VMs at presentation tier or application tier. Meanwhile, the system may stop VMs to save energy when the arrival flow reduces. We expect the algorithm to provide a relative coarse-grained resource scaling and complements the work in Chapter 5.

7.1.3 Predictive Server Consolidation in Multi-cores

We will extend the methodology presented in Chapter 6 to be able to accurately predict virtual machine performance interference on multi-cores. Rather than considering a single performance target for both applications, we will consider two heterogeneous applications, a primary and a secondary one, and answer the following

questions. Given a fixed number of VM instances and target response time for the primary application, what is the maximum consolidated instances of the secondary application so that the target response time of the primary application is not violated. To profile the application online, we will develop fine-grained system utility tools that can correctly monitor and store various resource utilization at the both hypervisor and virtual machine levels. Moreover, we should first develop fine-grained application monitoring tools that watch the network traffic and compute response time at each tier of web-based application. We expect that the profiling results will require the development of a load dependent queuing model to capture the relationship. Models developed in [50] will be readily transferred to capture consolidation of multi-tiered applications on multi-cores. In addition, we will aim to build a prediction tool that provides performance prediction in real time.

7.1.4 Server Consolidation with Performance Target

In Chapter 6, we developed a methodology to optimize total performance when placing m virtual machines on n servers. However, in some cases, applications service level objectives (SLOs) are required to be considered. Given a set of applications with their defined SLOs, it is not clear what is the number of servers needed and what is the application placement strategy so that *all* application targets can be met. Different SLO requirements require different amount of system resources for each application. We need to profile running applications, build their performance models online and figure out how much resources are needed to achieve their targets. Applications whose SLO violations are continuously detected are candidates to perform live migrations so that their performance can be improved. In contrast, applications whose measured performance is significantly better than the defined targets are candidates to consolidate in order to reduce the number of running servers. Models developed in Chapter 6 should guide the process of consolidation

or migration to avoid further SLO violations. Such a system could help applications achieve their SLOs and at the same time reduce the number of running servers.

A Markovian Arrival Processes

Markovian Arrival Processes (MAPs) can be seen as a generalization of continuous-time Markov chains (CTMC) used for fitting workload traces. A major difference between CTMC and MAPs is that transactions between states are classified by either as *background* transitions or as *completion* transitions. The *background* transitions only change the active state in the CTMC while *completion* transitions change the active state and conventionally trigger an arrival event. An inter-arrival time sample ΔT_k of a MAP model is the time between successive activation of any two completion transitions.

The commonly used MAP representation is the (D_0, D_1) description. If MAP has an infinitesimal generator Q of order N , the (D_0, D_1) representation is obtained by decomposing the transitions of Q according to whether or not it leads to a *completion* transition. D_0 has the same diagonal as Q but its off-diagonal elements are the rates of *background* transition; D_1 includes only the rates of *completion* transitions. It can be immediately computed that $Q = D_0 + D_1$.

For instance, a two-phase MAP may be specified as,

$$D_0 = \begin{bmatrix} -\lambda_{1,1} & \lambda_{1,2} \\ \lambda_{2,1} & -\lambda_{2,2} \end{bmatrix}, D_1 = \begin{bmatrix} \mu_{1,1} & \mu_{1,2} \\ \mu_{2,1} & \mu_{2,2} \end{bmatrix}$$

where $\lambda_{1,1} = \lambda_{1,2} + \mu_{1,1} + \mu_{1,2}$ and inverse of $\lambda_{1,1}$ is the mean time spent in phase 1 before a jump to other states. The off-diagonal elements $\lambda_{i,j}$, $i \neq j$, are the rates of

background transition. Similarly, elements of $\mu_{i,j}$, $i \neq j$, are the rates of following a *completion* transition.

The pseudo-code for reading the MAP configuration file and leveraging the MAP to generate random variates is shown in Algorithms A.1 and A.2. We define a new data structure STATE to help to store the MAP information as follows:

```
struct STATE{ double mean; double *p; };
```

Algorithm A.1 reads the D_0 and D_1 matrices from the configuration file and stores each state information into STATE structure. Algorithm A.2 generates an exponential random variate *interval* based on a global variable which stores the current state in MAP. Then it determines whether it performs a background or a completion transition based on a uniformly generated variable. If it is a completion transition, the algorithm outputs the *interval* as result; otherwise, recursively call `getInterval()` until an completion transition happens. The parameters for the three MAPs that generate the three burst levels used in the paper are as follows:

$$\text{Burst level 1: } D_0 = \begin{bmatrix} -7.572 & 0.0715 \\ 0.0692 & -0.1899 \end{bmatrix}, D_1 = \begin{bmatrix} 7.500 & 0 \\ 0 & 0.1206 \end{bmatrix}$$

$$\text{Burst level 2: } D_0 = \begin{bmatrix} -0.0661 & 0 \\ 0 & -16.9363 \end{bmatrix}, D_1 = \begin{bmatrix} 0.0548 & 0.0113 \\ 0.0406 & 16.8958 \end{bmatrix}$$

$$\text{Burst level 3: } D_0 = \begin{bmatrix} -0.0661 & 0 \\ 0 & -16.9363 \end{bmatrix}, D_1 = \begin{bmatrix} 0.0605 & 0.0057 \\ 0.0203 & 16.916 \end{bmatrix}$$

Algorithm A.1: readMAP() -- Read the MAP configuration file: build the D_0 and D_1 .

Input : MAP config file -- *input*

begin

```

    numState  $\leftarrow$  0 ;
    read (input, &numState);
    states  $\leftarrow$  new struct STATE [numState];
    for  $i \leftarrow$  numState do
        states[i].mean  $\leftarrow$  0;
        states[i].p  $\leftarrow$  new double[2 * numState];
    for ind  $\leftarrow$  0 to 1 do
        for  $i \leftarrow$  0 to numState - 1 do
            for  $j \leftarrow$  0 to numState - 1 do
                read (input, &states[i].p[ind * numState + j]);
                if states[i].p[ind * numState + j] < 0 then
                    states[i].p[ind * numState + j] = 0;
                states[i].mean  $\leftarrow$  states[i].mean + states[i].p[ind * numState + j];
            for ind  $\leftarrow$  0 to 1 do
                for  $i \leftarrow$  0 to numState - 1 do
                    for  $j \leftarrow$  0 to numState - 1 do
                        states[i].p[ind * numState + j]  $\leftarrow$ 
                            states[i].p[ind * numState + j] / states[i].mean;
                        if ind * numState + j > 0 then
                            states[i].p[ind * numState + j]  $\leftarrow$  states[i].p[ind *
                                numState + j] + states[i].p[ind * numState + j - 1];

```

Algorithm A.2: getInterval() -- Generate MAP random variates.

Input : global variable state index -- *cur_ind*

begin

```

    mean  $\leftarrow$  states[cur_ind].mean;
    interval  $\leftarrow$  Exponential(1/mean);
    prob  $\leftarrow$  Uniform(0, 1);
    for  $i \leftarrow$  0 to 2 * numState do
        if prob  $\leq$  states[cur_ind].p[i] then break;
    complete_idx  $\leftarrow$   $\lfloor \frac{i}{\text{numState}} \rfloor$ ;
    cur_ind  $\leftarrow$  i mod numState;
    if complete_idx = 0 then
        interval  $\leftarrow$  interval + getInterval();
    return interval;

```

References

- [1] Apache Software Foundation. <http://www.apache.org>.
- [2] Client-server model. <https://en.wikipedia.org/wiki/Client>
- [3] Credit Scheduler - Xen. http://wiki.xen.org/wiki/Credit_Scheduler.
- [4] Earliest deadline first scheduling.
http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.
- [5] Hyperic. <http://www.hyperic.com/>.
- [6] IEEE Spectrum: Tech titans building boom.
<http://www.spectrum.ieee.org/green-tech/buildings/tech-titans-building-boom>
- [7] iostat. <http://linux.die.net/man/1/iostat>.
- [8] IOzone Filesystem Benchmark. <http://www.iozone.org>.
- [9] Kernel based virtual machine. <http://www.linux-kvm.org/>.
- [10] Microsoft hyper-v. <http://www.microsoft.com/hyper-v-server>.
- [11] MongoDB. <http://www.mongodb.org>.
- [12] Netperf. <http://www.netperf.org/netperf/>.
- [13] Overview of Enterprise Applications.
<http://docs.oracle.com/javaee/5/firstcup/doc/gcrky.html>.

- [14] The qcow image format. <http://www.gnome.org/markmc/qcow-image-format.html>.
- [15] Standard Performance Evaluation Corporation.
<http://www.spec.org/cpu2006/>.
- [16] SysBench: a system performance benchmark.
<http://sysbench.sourceforge.net/>.
- [17] tcprstat. <http://launchpad.net/tcprstat>.
- [18] The Linux sar command. <http://linux.die.net/man/1/sar>.
- [19] Virtual Security In The Data Center.
<https://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns376/cisco-forrester-tap.pdf>.
- [20] VMware ESX and VMware ESXi.
<http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf>.
- [21] VMware ESX hypervisor. <http://www.vmware.com/products/vsphere/esxi-and-esx/index.html>.
- [22] VMware vSphere 5 Network I/O Control.
<http://www.vmware.com/products/datacenter-virtualization/vsphere/network-io-control.html>.
- [23] VMware vSphere. <http://www.vmware.com/products/vsphere/overview.html>.
- [24] VMware vSphere Web Services SDK.
<https://www.vmware.com/support/developer/vc-sdk/>.
- [25] Wikipedia. <http://en.wikipedia.org/wiki/Datacenter>.
- [26] Windows Hyper-V Server. <http://www.microsoft.com/hyper-v-server/>.

- [27] Citrix workload balancing 2.1 administrator's guide. *Citrix Whitepaper*, september 2011.
- [28] Tarek F. Abdelzaher and Nina Bhatti. Web Content Adaptation to Improve Server Overload Behavior. *Computer Network.*, 31:1563--1577, May 1999.
- [29] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Trans. Parallel Distrib. Syst.*, 13:80--96, January 2002.
- [30] Jussara Almeida, Mihaela Dabu, and Pei Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *Proc. of SIGMETRICS Workshop on Internet Server Performance*, pages 91--102, 1997.
- [31] Danilo Ansaloni, Lydia Y Chen, Evgenia Smirni, and Walter Binder. Model-driven consolidation of java workloads on multicores. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1--12. IEEE, 2012.
- [32] Simonetta Balsamo, Raif O. Onvural, and Vittoria De Nitto Persone'. *Analysis of Queueing Networks with Blocking*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
- [33] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45--58, 1999.
- [34] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles, SOSP '03*, pages 164--177, New York, NY, USA, 2003. ACM.

- [35] Novella Bartolini, Giancarlo Bongiovanni, and Simone Silvestri. An automatic admission control policy for distributed web systems. pages 138--144, 2007.
- [36] Novella Bartolini, Giancarlo Bongiovanni, and Simone Silvestri. Self-* through self-learning: overload control for distributed web systems. *Computer Networks*, 53:727--743, April 2009.
- [37] Aaron Beitch, Brandon Liu, Timothy Yung, Rean Griffith, Armando Fox, and David Patterson. Rain: A workload generation toolkit for cloud computing applications. In *U.C. Berkeley Technical Publications*, 2010.
- [38] Yigal Bejerano, Seung-Jae Han, and Li Erran Li. Fairness and load balancing in wireless lans using association control. pages 315--329, 2004.
- [39] Jean-Pascal Billaud and Ajay Gulati. hclock: Hierarchical qos for packet scheduling in a hypervisor. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 309--322, New York, NY, USA, 2013. ACM.
- [40] Peter Bodik, Rean Griffith, Charles Sutton, Armando Fox, Michael Jordan, and David Patterson. Statistical machine learning makes automatic control practical for internet datacenters. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, pages 12--16, 2009.
- [41] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412--447, November 1997.
- [42] Jakob Carlström and Raphael Rom. Application-aware Admission Control and Scheduling in Web Servers. In *21st Annual Joint Conference of the IEEE*

Computer and Communications Societies, INFOCOM '02, pages 506--515, 2002.

- [43] Giuliano Casale, Ningfang Mi, Ludmila Cherkasova, and Evgenia Smirni. Dealing with burstiness in multi-tier applications: Models and their parameterization. *IEEE Trans. Software Eng.*, 38(5):1040--1053, 2012.
- [44] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 242--261, New York, NY, USA, 2003. Springer-Verlag New York, Inc.
- [45] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 246--261, New York, NY, USA, 2002. ACM.
- [46] Surendar Chandra, Carla Schlatter Ellis, and Amin Vahdat. Differentiated Multimedia Web Services Using Quality Aware Transcoding. In *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, INFOCOM '00, pages 961--969, 2000.
- [47] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 103--116. ACM, 2001.
- [48] Chandra Chekuri and Sanjeev Khanna. On multi-dimensional packing problems. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete*

algorithms, SODA '99, pages 185--194, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

- [49] Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. Modeling program resource demand using inherent program characteristics. *SIGMETRICS Perform. Eval. Rev.*, 39:1--12, June 2011.
- [50] Lydia Y Chen, Danilo Ansaloni, Evgenia Smirni, Akira Yokokawa, and Walter Binder. Achieving application-centric performance targets via consolidation on multicores: myth or reality? In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 37--48. ACM, 2012.
- [51] Xiangping Chen, Prasant Mohapatra, and Huamin Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proceedings of the 10th international conference on World Wide Web, WWW '01*, pages 545--554, 2001.
- [52] Ludmila Cherkasova. Scheduling Strategy to Improve Response Time for Web Applications. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe 1998*, pages 305--314, 1998.
- [53] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42--51, September 2007.
- [54] Ludmila Cherkasova and Peter Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Trans. Comput.*, 51:669--685, June 2002.

- [55] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273--286. USENIX Association, 2005.
- [56] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483--490, September 1981.
- [57] Mark E. Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection Scheduling in Web Servers. In *Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems - Volume 2*, pages 22--33, 1999.
- [58] N.R. Draper and H. Smith. *Applied Regression Analysis*. John Wiley and Sons, Third Edition, 1998.
- [59] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM symposium on Operating systems principles, SOSP '99*, pages 261--276, New York, NY, USA, 1999. ACM.
- [60] Lars Eggert and John Heidemann. Application-level Differentiated Services for Web Servers. *World Wide Web*, 2:133--142, March 1999.
- [61] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the 13th international conference on World Wide Web, WWW '04*, pages 276--286, 2004.
- [62] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of*

the sixteenth ACM symposium on Operating systems principles, SOSP '97, pages 78--91, 1997.

- [63] N Gandhi, DM Tilbury, Y Diao, J Hellerstein, and S Parekh. MIMO control of an apache web server: Modeling and controller design. In *American Control Conference, 2002. Proceedings of the 2002*, volume 6, pages 4922--4927. IEEE, 2002.
- [64] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation, NSDI'11*, pages 24--37, Berkeley, CA, USA, 2011. USENIX Association.
- [65] Daniel Gmach, Jerry Rola, Ludmila Cherkasova, and Alfons Kemper. Capacity management and demand prediction for next generation data centers. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 43--50. IEEE, 2007.
- [66] Ashish Goel, Adam Meyerson, and Serge Plotkin. Approximate majorization and fair online load balancing. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 384--390, 2000.
- [67] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11*, pages 22:1--22:14, New York, NY, USA, 2011. ACM.
- [68] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *Proceedings of the*

7th conference on File and storage technologies, FAST '09, pages 85--98, Berkeley, CA, USA, 2009. USENIX Association.

- [69] Ajay Gulati, Anne Holler, Minwen Ji, Ganesha Shanmuganathan, Carl Waldspurger, and Xiaoyun Zhu. Vmware distributed resource management: Design, implementation, and lessons learned. *VMware Technical Journal*, 1(1):45--64, 2012.
- [70] Ajay Gulati, Arif Merchant, and Peter J. Varman. mclock: Handling throughput variability for hypervisor IO scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 437--450. USENIX Association, 2010.
- [71] Ajay Gulati, Ganesha Shanmuganathan, Xuechen Zhang, and Peter Varman. Demand based hierarchical qos using storage resource pools. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference, USENIX ATC'12*, pages 1--13, Berkeley, CA, USA, 2012. USENIX Association.
- [72] Diwaker Gupta, Ludmila Cherkasova, Rob Gardner, and Amin Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Middleware '06*, pages 342--362, New York, NY, USA, 2006. Springer-Verlag New York, Inc.
- [73] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. Xenmon: Qos monitoring and performance profiling tool. Technical report, 2005.
- [74] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-Interscience, 2004.

- [75] V.M. Infrastructure. Resource management with vmware drs. *VMware Whitepaper*, 2006.
- [76] Canturk Isci, James E Hanson, Ian Whalley, Malgorzata Steinder, and Jeffrey O Kephart. Runtime demand estimation for effective dynamic resource management. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 381--388. IEEE, 2010.
- [77] Ravi Iyer, Ramesh Illikkal, Omesh Tickoo, Li Zhao, Padma Apparao, and Don Newell. Vm3: Measuring, modeling and managing vm shared resources. *Comput. Netw.*, 53:2873--2887, December 2009.
- [78] Ravi Iyer, Vijay Tewari, and Krishna Kant. Overload Control Mechanisms for Web Servers. In *Workshop on Performance and QoS of Next Generation Networks*, pages 225--244, 2000.
- [79] Wei Jin, Jeffrey S Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 37--48. ACM, 2004.
- [80] Vikram Kanodia and Edward W. Knightly. Ensuring Latency Targets in Multiclass Web Servers. *IEEE Trans. Parallel Distrib. Syst.*, 14:84--93, January 2003.
- [81] Christos Karamanolis, Magnus Karlsson, and Xiaoyun Zhu. Designing controllable computer systems. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10, HOTOS'05*, pages 9--9, Berkeley, CA, USA, 2005. USENIX Association.
- [82] Jon Kleinberg, Yuval Rabani, and Éva Tardos. Fairness in routing and load balancing. In *J. Comput. Syst. Sci*, pages 568--578, 1999.

- [83] V.M. Koch. *A factor graph approach to model-based signal separation*. Hartung-Gorre Verlag, 2007.
- [84] Palden Lama and Xiaobo Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS '10*, pages 151--160, Washington, DC, USA, 2010. IEEE Computer Society.
- [85] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research Technical Report*, pages MSR--TR--2011--9, january 2011.
- [86] Kelvin Li and Sugih Jamin. A Measurement-Based Admission-Controlled Web Server. In *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM '00*, pages 651--659, 2000.
- [87] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference, ATEC '06*, pages 29--42, Berkeley, CA, USA, 2006. USENIX Association.
- [88] H-A Loeliger. An introduction to factor graphs. *Signal Processing Magazine, IEEE*, 21(1):28--41, 2004.
- [89] Chenyang Lu, Tarek F. Abdelzaher, John A. Stankovic, and Sang H. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. In *Proceedings of the Seventh Real-Time Technology and Applications Symposium, RTAS '01*, pages 51--62, 2001.

- [90] Lei Lu, Ludmila Cherkasova, Vittoria de Nitto Personè, Ningfang Mi, and Evgenia Smirni. Await: Efficient overload management for busy multi-tier web services under bursty workloads. In *Proceedings of the 10th international conference on Web engineering, ICWE'10*, pages 81--97, Berlin, Heidelberg, 2010. Springer-Verlag.
- [91] Lei Lu, Hui Zhang, Guofei Jiang, Haifeng Chen, Kenji Yoshihira, and Evgenia Smirni. Untangling mixed information to calibrate resource utilization in virtual machines. In *Proceedings of the 8th ACM international conference on Autonomic computing, ICAC '11*, pages 151--160, New York, NY, USA, 2011. ACM.
- [92] Lei Lu, Hui Zhang, Evgenia Smirni, Guofei Jiang, and Kenji Yoshihira. Predictive vm consolidation on multiple resources: Beyond load balancing. In *the 21st IEEE/ACM International Symposium on Quality of Service (IWQoS'13)*, pages 1--10, June 2013.
- [93] Lei Lu, Xiaoyun Zhu, Rean Griffith, Pradeep Padala, Aashish Parikh, Parth Shah, and Evgenia Smirni. Application-driven auto-scaling of virtual machines in resource pools. In *Network Operations and Management Symposium, NOMS '14*. IEEE, 2014, submitted for publication.
- [94] Ying Lu, Tarek F Abdelzaher, and Avneesh Saxena. Design, implementation, and evaluation of differentiated caching services. *Parallel and Distributed Systems, IEEE Transactions on*, 15(5):440--452, 2004.
- [95] Morris L Marx and Richard J Larsen. *Introduction to mathematical statistics and its applications*. Pearson/Prentice Hall, 2006.
- [96] Marvin McNett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker. Usher: An Extensible Framework for Managing Clusters of Virtual Machines.

In *Proceedings of the 21st Large Installation System Administration Conference (LISA)*, pages 1--15, Berkeley, CA, USA, November 2007. USENIX Association.

- [97] Daniel A Menascé, Virgílio AF Almeida, and Larry W Dowdy. *Capacity planning and performance modeling: from mainframes to client-server systems*. Prentice-Hall, Inc., 1994.
- [98] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Burstiness in Multi-tier Applications: Symptoms, Causes, and New Models. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 265--286, 2008.
- [99] Ningfang Mi, Giuliano Casale, Ludmila Cherkasova, and Evgenia Smirni. Injecting Realistic Burstiness to a Traditional Client-Server Benchmark. In *Proceedings of the 6th international conference on Autonomic computing, ICAC '09*, pages 149--158, 2009.
- [100] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems, EuroSys '10*, pages 237--250, New York, NY, USA, 2010. ACM.
- [101] Giovanni Pacifici, Wolfgang Segmuller, Mike Spreitzer, and Asser Tantawi. Dynamic estimation of cpu demand of web traffic. In *Proceedings of the 1st international conference on Performance evaluation methodologies and tools, valuetools '06*, New York, NY, USA, 2006. ACM.
- [102] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European con-*

ference on Computer systems, EuroSys '09, pages 13--26, New York, NY, USA, 2009. ACM.

- [103] Sujay Parekh, Neha Gandhi, Joseph Hellerstein, Dawn Tilbury, T Jayram, and Joe Bigus. Using control theory to achieve service level objectives in performance management. *Real-Time Systems*, 23(1-2):127--141, 2002.
- [104] Harry G. Perros. *Queueing Networks with Blocking*. Oxford University Press, Inc., New York, NY, USA, 1994.
- [105] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412--421, July 1974.
- [106] John Scott Robin and Cynthia E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th conference on USENIX Security Symposium - Volume 9*, SSYM'00, pages 10--25, Berkeley, CA, USA, 2000. USENIX Association.
- [107] Jerry Rolia, Ludmila Cherkasova, Martin Arlitt, and Artur Andrzejak. A capacity management service for resource pools. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 229--237, New York, NY, USA, 2005. ACM.
- [108] Kenneth H Rosen and Kamala Krithivasan. *Discrete mathematics and its applications*, volume 6. McGraw-Hill New York, 1999.
- [109] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 29--42, Berkeley, CA, USA, 2008. USENIX Association.

- [110] Stefan Seltzsam, Daniel Gmach, Stefan Krompass, and Alfons Kemper. Autoglobe: An automatic administration concept for service-oriented database applications. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE '06*, pages 90--101, Washington, DC, USA, 2006. IEEE Computer Society.
- [111] Kai Shen, Hong Tang, Tao Yang, and Lingkun Chu. Integrated Resource Management for Cluster-based Internet Services. *SIGOPS Oper. Syst. Rev.*, 36:225--238, December 2002.
- [112] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [113] Preeti Bhoj Srinivas, Srinivas Ramanathan, and Sharad Singhal. Web2K: Bringing QoS to Web Servers. Technical Report HPL-2000-61, HPLabs, 2000.
- [114] L. Surhone, M Timpledon, and S Marseken. *Source Separation: Digital Signal Processing, Signal (Electronics), Principal Component Analysis, Independent Component Analysis, Auditory Scene Analysis, Magnetoencephalography*. Betascript Publishers, 2010.
- [115] Chris Takemura and Luke Seidel Crawford. *The Book of Xen: A Practical Guide for the System Administrator*. No Starch Press, 2010.
- [116] Leandros Tassiulas and Saswati Sarkar. Maxmin fair scheduling in wireless networks. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 763--772. IEEE, 2002.

- [117] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Trans. Auton. Adapt. Syst.*, 3(1):1:1--1:39, March 2008.
- [118] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Oper. Syst. Rev.*, 36(SI):239--254, dec 2002.
- [119] VMware, Inc. *vSphere Resource Management Guide: ESXi 5.1, vCenter Server 5.1*. 2012.
- [120] Werner Vogels. Beyond server consolidation. *Queue*, 6(1):20--26, January 2008.
- [121] Thiemo Voigt. Overload behaviour and protection of event-driven web servers. In *Web Engineering and Peer-to-Peer Computing*, pages 147--157. Springer, 2002.
- [122] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 189--202, 2001.
- [123] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181--194, 2002.
- [124] Zhikui Wang, Yuan Chen, Daniel Gmach, Sharad Singhal, Brian J. Watson, Wilson Rivera, Xiaoyun Zhu, and Chris Hyser. Appraise: application-level performance management in virtualized server environments. *IEEE Trans. on Network and Service Management*, 6(4):240--254, 2009.

- [125] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230--243, 2001.
- [126] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. Scale and performance in the denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195--209, 2002.
- [127] Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 366--387, New York, NY, USA, 2008. Springer-Verlag New York, Inc.
- [128] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 17--30, Berkeley, CA, USA, 2007. USENIX Association.
- [129] Jing Xu, Ming Zhao, José Fortes, Robert Carpenter, and Mazin Yousif. Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. *Cluster Computing*, 11(3):213--227, 2008.
- [130] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 27--36, Washington, DC, USA, 2007. IEEE Computer Society.

- [131] Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West. Friendly virtual machines: leveraging a feedback-control model for application adaptation. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 2--12. ACM, 2005.

VITA

Lei Lu received his B.S. degree in Computer Science from Nanjing University, China, in 2005 and his M.E. degree in System Engineering from Nanjing University, China, in 2008. He has been a Ph.D. candidate in Computer Science Department at the College of William and Mary since 2009. His research interests include server virtualization, system performance evaluation, and cloud computing.